ST7
8-BIT MCU FAMILY

USER GUIDE

July 2002

USE IN LIFE SUPPORT DEVICES OR SYSTEMS MUST BE EXPRESSLY AUTHORIZED.

STMicroelectronics PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF STMicroelectronics. As used herein:

1. Life support devices or systems are those which (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided with the product, can be reasonably expected to result in significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can reasonably be expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

Table of Contents

1 INTRODUCTION 12

1.1 WHO IS THIS BOOK WRITTEN FOR? 12

1.2 ABOUT THE AUTHORS 12

1.3 HOW IS THIS BOOK ORGANIZED? 12

1.4 WHY A MICROCONTROLLER? 13

1.4.1 Electronic circuitry 15
1.4.2 Choice of microcontroller model 17
1.4.3 Choice of development tools 17

2 HOW DOES A TYPICAL MICROCONTROLLER WORK? 19

2.1 THE CENTRAL PROCESSING UNIT 20

2.2 HOW THE CPU AND ITS PERIPHERALS MAKE UP A SYSTEM 21

2.2.1 CPU 21
2.2.2 Memory 21
2.2.3 Input-Outputs 23
2.2.4 Interrupt Controller 24
2.2.5 Bus 25
2.2.6 Clock Generator 25
2.2.7 Reset Generator 25

2.3 CORE 25

2.3.1 Arithmetic and Logic Unit (ALU) 25
2.3.2 Program Counter 26
2.3.3 Instruction Decoder 26
2.3.4 Stack Pointer 26

2.4 PERIPHERALS 27

2.4.1 Parallel Input-Outputs 27
2.4.2 Analog to Digital Converter 28
2.4.3 Programmable Timer 28
2.4.4 Serial Peripheral Interface 28
2.4.5 Watchdog Timer 28

2.5 THE INTERRUPT MECHANISM AND HOW TO USE IT 29

2.5.1 Interrupt handling 29
2.5.1.1 Hardware mechanism 31
2.5.1.2 Hardware sources of interrupt 31
2.5.1.3 Global interrupt enable bit 32

Table of Contents

2.5.1.4 Software interrupt instruction 32
2.5.1.5 Saving the state of the interrupted program 32
2.5.1.6 Interrupt vectorization 32
2.5.1.7 Interrupt service routine 34
2.5.1.8 Interrupt Return instruction 34
2.5.2 Software precautions related to interrupt service routines 34
2.5.2.1 Saving the Y register 34
2.5.2.2 Managing the stack 35
2.5.2.3 Resetting the hardware interrupt request flags 35
2.5.2.4 Making an interrupt service routine interruptible 35
2.5.2.5 Data desynchronization and atomicity 36
2.5.3 Conclusion: the benefits of interrupts 38

2.6 AN APPLICATION USING INTERRUPTS: A MULTITASKING KERNEL 39
2.6.1 Pre-emptive multitasking 39
2.6.2 Cooperative multitasking 41
2.6.3 Multitasking kernels 42
2.6.3.1 Advantages of programming with a multitasking kernel 42
2.6.3.2 The task declaration and allocation 42
2.6.3.3 Task sleeping and waking-up 42
2.6.3.4 Multitasking kernel overhead 43

3 PROGRAMMING A MICROCONTROLLER 45

3.1 ASSEMBLY LANGUAGE 45
3.1.1 When to use assembly language 45
3.1.2 Development process in assembly language 46
3.1.2.1 Assembly language 47
3.1.2.2 Assembler 48
3.1.2.3 Linker 49
3.1.2.4 The project builder/make utility 51
3.1.2.5 EPROM burners 52
3.1.2.6 Simulators 53
3.1.2.7 In-circuit emulators 54

3.2 C LANGUAGE 55
3.2.1 Why use C? 55
3.2.2 Tools used with C language 57
3.2.3 Debugging in C 58

3.3 DEVELOPMENT CHAIN SUMMARY 60

3.4 APPLICATION BUILDERS 61

3.5 FUZZY-LOGIC COMPILERS 61

Table of Contents

4 ARCHITECTURE OF THE ST7 CORE 62

4.1 POSITION OF THE ST7 WITHIN THE ST MCU FAMILY 62

4.2 ST7 CORE 63

4.2.1 Addressing space 65

4.2.2 Internal registers 65

4.2.2.1 Accumulator (A) 65

4.2.2.2 Condition Code register (CC) 65

4.2.2.3 Index registers (X and Y) 67

4.2.2.4 Program Counter (PC) 68

4.2.2.5 Stack Pointer (SP) 68

4.3 INSTRUCTION SET AND ADDRESSING MODES 70

4.3.1 A word about mnemonic language 70

4.3.2 Addressing modes 72

4.3.3 Instruction set 73

4.3.4 Coding of the instructions and the address 74

4.3.4.1 Prefix byte 74

4.3.4.2 Opcode byte 75

4.3.4.3 The addressing modes in detail 77

4.4 ADVANTAGES OF THE ST7 INSTRUCTION SET AND ADDRESSING MODES 82

5 PERIPHERALS 84

5.1 CLOCK GENERATOR 84

5.1.1 ST72251 Miscellaneous Register 84

5.1.2 ST72311 Miscellaneous Register 85

5.2 INTERRUPT PROCESSING 86

5.2.1 Interrupt sources and interrupt vectors 86

5.2.1.1 Interrupts sources for the ST72251 87

5.2.1.2 Interrupt sources for the ST72311 88

5.2.2 Interrupt vectorization 89

5.2.3 Global interrupt enable bit 90

5.2.4 TRAP instruction 91

5.2.5 Interrupt mechanism 91

5.2.5.1 Saving the interrupted program state 91

5.2.5.2 Interrupt service routine 91

5.2.5.3 Restoring the interrupted program state: The IRET instruction 92

5.2.6 Nesting the interrupt services 92

5.3 PARALLEL INPUT-OUTPUT PORTS 94

Table of Contents

5.3.1 ST72251 I/O Ports 94
5.3.2 ST72311 I/O Ports 96

5.4 WATCHDOG TIMER 99
5.4.1 Aim of the watchdog 99
5.4.2 Watchdog Description 100
5.4.3 Using the Watchdog to protect an application 103

5.5 16-BIT TIMER 103
5.5.1 Timer clock 104
5.5.2 Free running counter 105
5.5.2.1 Reading the free running counter 105
5.5.2.2 Resetting the free running counter 106
5.5.2.3 The TOF flag 107
5.5.3 Input capture operation 108
5.5.4 Output compare operation 110
5.5.5 One-pulse mode 113
5.5.6 Pulse-Width Modulation mode 115

5.6 ANALOG TO DIGITAL CONVERTER 117
5.6.1 Description 117
5.6.2 Using the Analog to Digital Converter 118
5.6.3 The problem of the converter's accuracy 119
5.6.4 Using the ADC to convert positive and negative voltages; increasing its resolution . . 120
5.6.4.1 Measuring negative and positive voltages 120
5.6.4.2 Increasing the resolution 121
5.6.4.3 Application Examples 124

5.7 SERIAL PERIPHERAL INTERFACE 125

5.8 SERIAL COMMUNICATION INTERFACE 128
5.8.1 Bit rate generator 128
5.8.2 Send and receive mechanism 129
5.8.3 Status register 132
5.8.4 Control Register 2 132
5.8.5 Using the Wake-Up feature in a multiprocessor system 133
5.8.6 Handling the interrupts 133

6 STMICROELECTRONICS PROGRAMMING TOOLS 135

6.1 ASSEMBLER 135
6.1.1 An overview of the assembler function 135
6.1.2 Instruction coding 137

Table of Contents

6.1.3 Declaring variables 138
6.1.4 Declaring constants 140
6.1.4.1 Constant data 140
6.1.4.2 Symbol definition 141
6.1.5 Relocation commands 142
6.1.5.1 What is relocation? 142
6.1.5.2 Segment definition 143
6.1.5.3 Using the Segment directive in the source file 145
6.1.5.4 Segment allocation 146
6.1.5.5 Initialization of variables at power-on 148
6.1.5.6 Referencing symbols and labels between modules 151
6.1.6 Conditional assembly 154
6.1.7 Macros 156
6.1.7.1 Replaceable parameters 157
6.1.7.2 Local symbols 158
6.1.7.3 Conditional statements in macros 160
6.1.8 Some miscellaneous features 162
6.1.8.1 EQU and CEQU pseudo-ops 162
6.1.8.2 #DEFINE pseudo-op 162
6.1.8.3 Numbering syntax directives 163
6.1.9 Object and listing files 163
6.1.9.1 Object files 163
6.1.9.2 Listing files 164

6.2 LINKER AND ASCII-HEX CONVERTER 165
6.2.1 The linking process 165
6.2.2 Hex file translator 167
6.2.3 The back-annotation pass of the assembler 168

6.3 INSTALLING WINEDIT AND THE SOFTWARE TOOLS 168
6.3.1 WinEdit text editor 168
6.3.1.1 Installing WinEdit 168
6.3.1.2 Configuring WinEdit 169
6.3.2 Installing the STMicroelectronics Software Tools 169

6.4 BUILDING A DEMONSTRATION PROGRAM 170
6.4.1 Purpose of the demonstration program 170
6.4.2 Inventory of the program files 170
6.4.3 Description of the program files 171
6.4.3.1 The PROJECT.WPJ file 171
6.4.3.2 The main source file, MAIN.ASM and the timer source file, TIMER500.ASM . . . 173
6.4.3.3 The REG72251.ASM file and the REGISTER.INC file 176
6.4.3.4 The MAP72251.ASM file 178

Table of Contents

6.4.3.5 The CATERPIL.BAT file 179
6.4.4 Using WinEdit to change and compile the files 180

7 DEBUGGER AND PROM PROGRAMMER TUTORIAL FOR ST72251 183

7.1 STMICROELECTRONICS HARDWARE TOOLS 183
7.1.1 EPROM Programming Boards 183
7.1.2 Starter Kits 184
7.1.3 Development Kits 184
7.1.4 Emulators 184

7.2 EPROM PROGRAMMER BOARDS 184
7.2.1 EPROM programmer Installation 185
7.2.2 Using the EPROMER software 185

7.3 EMULATOR AND DEBUGGER 189
7.3.1 Introducing the emulator and the debugger 189
7.3.2 Installing the emulator and the debugger 189
7.3.3 Using the debugger 191
7.3.3.1 Loading the application 191
7.3.3.2 Running the application 193
7.3.3.3 Watching the registers and variables 193
7.3.3.4 Using Inspect and Watch 195
7.3.3.5 Using breakpoints 197
7.3.3.6 Watching the contents of the stack 198
7.3.3.7 Watching the execution trace 199
7.3.3.8 More features to come later 200

7.4 PURPOSE OF THE TUTORIAL 200

7.5 SCHEMATIC DRAWING OF THE PRINTED CIRCUIT BOARD 202

7.6 DEVELOPING THE PROGRAM 202
7.6.1 Peripherals used to implement the solution 202
7.6.2 The algorithm of each task 203
7.6.3 A simple multitasking kernel for the ST7 204
7.6.3.1 StartTasks routine 204
7.6.3.2 The Yield routine 206
7.6.4 The source code of the application 209
7.6.4.1 Main file (Multitsk.asm) 210
7.6.4.2 ADC source file(Acana.asm) 214
7.6.4.3 Kernel source file (Littlk.asm) 215

7.7 RUNNING THE APPLICATION 217

Table of Contents

7.8 SUMMARY REMARKS 217

8 C LANGUAGE AND THE C COMPILER 219

8.1 C LANGUAGE EXTENSIONS FOR MICROCONTROLLERS 219

8.2 DESCRIPTION AND INSTALLATION OF THE HICROSS TOOL CHAIN 220

8.3 USING THE C COMPILER 224

8.3.1 Memory allocation 224

8.3.1.1 Read-only constants 225

8.3.1.2 EEPROM non-volatile storage 226

8.3.1.3 Page Zero variables 227

8.3.1.4 Far and near pointers 227

8.3.2 Initialization of variables and constant variables 228

8.3.3 Inputs and outputs 228

8.3.3.1 First method: using macros 229

8.3.3.2 Second method: defining variables 229

8.3.4 Interrupt handling 230

8.3.5 Limitations put on the full implementation of C language 230

8.4 USING THE ASSEMBLER 231

8.4.1 Using In-line assembler statements within a C source text 231

8.4.1.1 Single-statement assembler block 231

8.4.1.2 Multiple-statement assembler block 232

8.4.2 Using the Hiware assembler 233

8.5 USING THE LINKER 233

8.6 USING THE EPROM BURNER 235

8.7 PROJECT DIRECTORY STRUCTURE 237

8.7.1 Config directory 237

8.7.2 Object directory 239

8.7.3 Sources directory 239

8.8 HINTS ON C WRITING STYLE FOR THE ST7 241

8.8.1 Accessing individual bits in registers 241

8.8.2 Setting configuration registers 243

8.8.3 Using macros to define external devices 243

8.8.4 Optimizing resource usage 245

8.8.4.1 Define a function when a group of statements is repeated several times 245

8.8.4.2 Use shifts instead of multiplication and division 245

8.8.4.3 Limit the size of variables to the very minimum 246

Table of Contents

8.9 CONCLUSION 246

9 A CARRIER-CURRENT SYSTEM FOR DOMESTIC REMOTE CONTROL 247

9.1 CARRIER CURRENT CONTROL AND THE X-10 STANDARD 248

9.2 TRANSMITTER 253

9.2.1 Instructions for use 253
9.2.2 Description of the electronic circuit 253
9.2.3 Description of the software 257
9.2.3.1 The main program 257
9.2.3.2 Timer A Capture interrupt service routine 260
9.2.3.3 The Timer B overflow interrupt service routine 267

9.3 RECEIVER 270

9.3.1 Instructions for use 270
9.3.2 Electronic circuitry 270
9.3.3 Software 274
9.3.3.1 Interrupt functions 274
9.3.3.2 Main program 277

9.4 CONCLUSION 283

10 SECOND APPLICATION: A SAILING COMPUTER 284

10.1 THEORY OF THE COMPUTATION 286

10.2 INTERFACING THE MEASUREMENT DEVICES 289

10.2.1 Frequency-type devices: speedometer and wind gauge 289
10.2.1.1 Interfacing the speedometer 289
10.2.1.2 Interfacing the wind gauge 289
10.2.1.3 Using a common timer for both speed measurement devices 290
10.2.2 Interfacing the weather vane 291

10.3 INTERFACING THE DISPLAY 292

10.3.1 Display circuit 293
10.3.2 Push-button circuit 296
10.3.3 LED circuit 297

10.4 INTERFACING THE OPTIONAL PERSONAL COMPUTER 297

10.5 PROGRAM ARCHITECTURE 298

10.5.1 Reading and conversion of the speeds 298
10.5.2 Refreshing of the display 300

10.5.3 Polling the push-buttons 302
10.5.4 Reading and filtering the wind direction 303
10.5.5 The periodic interrupt service routine 304
10.5.6 Computation of the results 305
10.5.7 Handling of the serial interface 307
10.5.8 Initialization of the peripherals and the parameters 308

10.6 MEMORY ALLOCATION AND COMPILE AND LINK OPTIONS 310

10.7 CONCLUSION 312

11 SOME LAST REMARKS 313

1 INTRODUCTION

1.1 WHO IS THIS BOOK WRITTEN FOR?

This book is a technical guide for ST7 users and may be approached in different ways:

- For students and anyone unfamiliar with microprocessors, but with some experience of logic circuits; they should start by reading Chapters 1 through 3.
- For trained engineers wanting to get specific knowledge about the ST7 and microcontroller programming in C language; they may skip Chapters 1 through 3 and go straight to Chapter 4.
- For designers already familiar with the ST7, needing more details about C-language programming and how to use the ST7 internal peripherals; the application descriptions in Chapters 5 and 8 through 10 are of special interest for them.

1.2 ABOUT THE AUTHORS

Jean-Luc Gregoriades teaches automated systems and industrial computer science at the Electrical Engineering department of the University of Cergy-Pontoise, France.

Jean-Marc Delaplace is an electronics and software engineer at Gilson S.A., a laboratory automation instrument maker.

As a team, they have already written books on the ST6 (published at Dunod Editions) and the ST9 (published by STMicroelectronics).

1.3 HOW IS THIS BOOK ORGANIZED?

This book contains the following chapters:

Chapter 1: Introduction.

Chapter 2: How does a typical microcontroller work internally and how to use it.

Chapter 3: Programming a microcontroller.

Chapter 4: Architecture of the ST7 core.

Chapter 5: The peripherals.

Chapter 6: The STMicroelectronics programming tools.

Chapter 7: The Debugger and the PROM programmer through a pedagogic application using a ST72251.

Chapter 8: The C language and the C compiler.

Chapter 9: Application of the ST72251: a carrier-current system for domestic remote control.

Chapter 10: Application of the ST72311: a sailing computer.

Chapter 11: Conclusion.

Chapters 1, 2 and 3 are a refresher on the concept of a microcontroller. Chapter 1 introduces the concept, Chapter 2 addresses the hardware and Chapter 3 addresses the software aspects.

Chapters 4 through 7 describe the ST7 and its programming tools, taking only assembly language into account.

Chapter 8 discusses the C language and techniques for using the C Compiler for the ST7 microcontroller, its strengths and also its limitations.

Chapters 9 and 10 describe application projects using the ST72251 and the ST72311 members of the ST7 family. They tell the story of the design of devices that, though they do work, were not intended to be commercial products.

1.4 WHY A MICROCONTROLLER?

The microcontroller is just another choice when one has to design an application, and it competes with other technologies, like wired logic, a microprocessor system, or a Programmable Logic Device of which many types are available.

All these solutions tend to reduce the number of components, the area of printed circuit used, the number of connections, while increasing the computing power and keeping the cost low.

The following table shows a comparison of these solutions. Each one is discussed below.

Solution type	Advantages	Drawbacks
Wired logic	Very high speed Cheap	Only for simple circuits
Programmable logic	High speed Able to handle complex digital signals	Limited number processing Programming languages are specific and non-portable May be expensive
Microprocessor	Powerful Wide choice of models Configurable in wide limits Allows almost all popular programming languages	Many components even for simple systems Relatively expensive
Microcontroller	Simple electronic circuits are possible with few components Allows the most popular programming languages such as BASIC or C.	Standard configurations rarely exactly fit the application's needs implying the use of over-sized models Special configurations available, but only for large quantities.

Wired logic uses commercially available logic functions and sometimes linear chips. Though it is simple, it is neither practical nor economical to consider this technology for building applications as complex as those that are usually needed today. It can only be considered for very special subfunctions where high speed is required.

Programmable Logic Devices (PLD) are the modern form of wired logic, and are often used for combinatory and sequential logic. The biggest models allow intensive numeric processing, but only on integer numbers. They use programming languages that do not belong to the family of computer languages commonly used today.

The last two technologies are the microprocessor and the microcontroller. In principle, both are very much alike and they are both well suited to programmed data processing. The main difference between them is the size of the application.

The microprocessor is a component that includes mainly the computing core, and perhaps the logic closely related to it like the clock generator, the interrupt controller, etc. Many more chips must be added to it in order to make a functional application, memory chips in particular. Actually, this solution is only used in computers, either general-purpose computers like PCs, or built-in to complex applications like industrial robots. It allows the designer to tailor his circuit exactly to his needs.

The microcontroller is defined as a complete programmed system in one chip. This means that one chip is sufficient to fulfil the need, or that only a few more chips are required to

achieve the required computational power. These external chips may simply be interface components, to adapt the electric signals to the input-output pins of the microcontroller, or additional memory or peripheral components if the buses are available externally on the pins of the microcontroller.

In any case, these components require two different but equally important jobs for putting them to work: electronic circuit design, and programming.

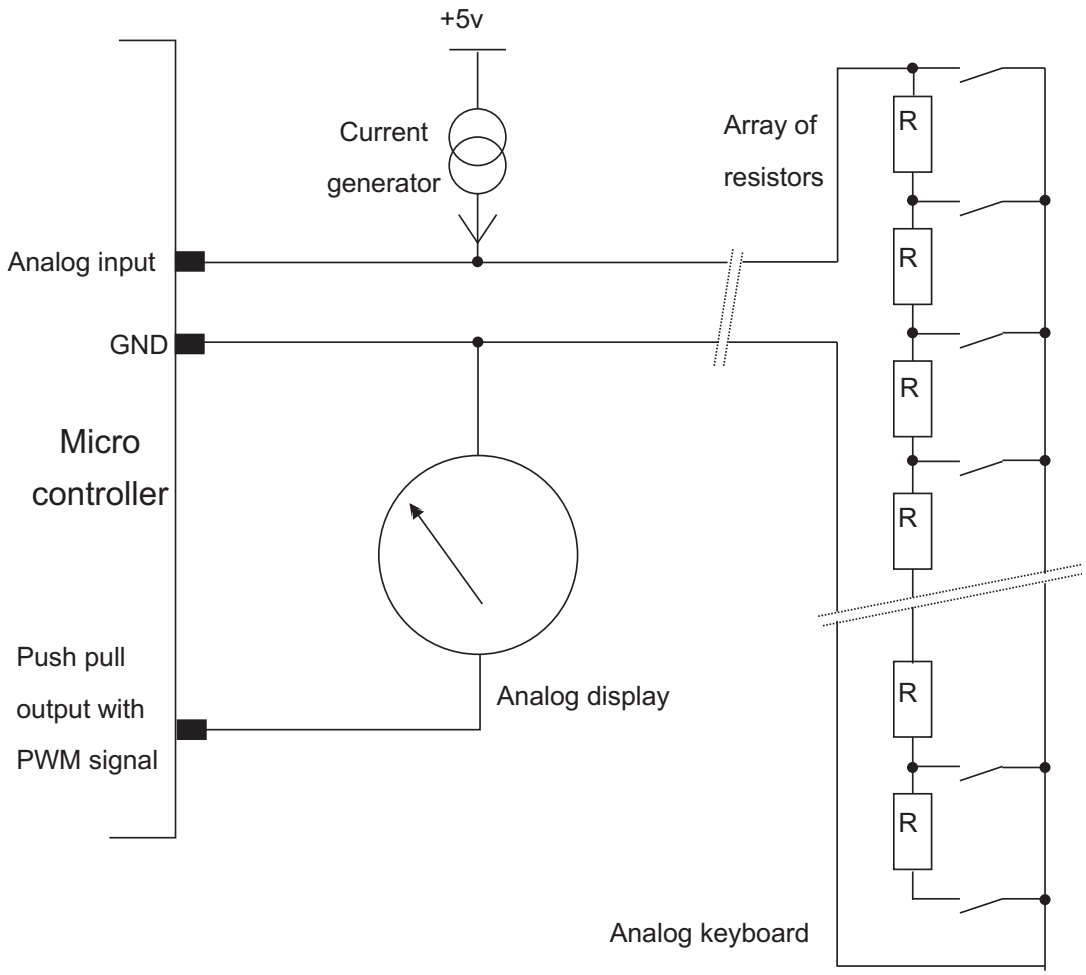
Both of these need be done as easily, quickly and economically as possible. A thorough study of both aspects will be the basis for selecting the most appropriate model from the wide range of products available today. Here are a few considerations related to these aspects.

1.4.1 Electronic circuitry

This is where the designer strives to reduce the external component count, and to carefully select each one to get the best value. In order to satisfy this requirement, the various chip manufacturers offer for each family a choice of variants, to allow the designer to select the one that best fits his needs in terms of input-outputs and auxiliary circuitry.

Roughly speaking, a microcontroller variant that is loaded with features will allow a simpler external circuitry, at the expense of an increase in the microcontroller cost. The ideal choice would be the variant that has the exact peripherals required by the application, and no more.

To illustrate this, we shall take a simple example. Let us consider an application that requires, as an input, a numeric keypad, and as an output, a galvanometer to provide an analog display. The ideal combination would call for an Analog to Digital Converter for the input, and a programmable timer with Pulse Width Modulation capability for the output. This would lead to the following very simple schematic:



Example of simplified circuitry using a microcontroller

01-anal

Such peripherals are typically available in many families. This example shows how two peripherals properly selected can drastically reduce the component count and thus the printed circuit area. The solution shown may or may not fit the needs, but it is difficult to imagine a simpler design.

1.4.2 Choice of microcontroller model

The selected model of microcontroller must meet the requirements in terms of computational power. It must be able to handle the input-outputs, process the data in the required amount of time, and have enough memory to store both the program and the data.

An application is made of both hardware and software. So, there is a trade-off between the processing done by hardware and that done by software. Using dumb peripherals requires more computational power from the core; using sophisticated peripherals relieves the core from time-consuming calculations and thus allows a less powerful core to be chosen.

Determining the computational power is a difficult matter since there is no internationally recognized measurement unit that expresses the speed of a microprocessor or similar device.

Some benchmarks that compare several products in the same application are available from various sources, but they only give an idea of the relative capability of one product versus another one.

Thus a certain margin must be considered, or there would be a risk that some time in the development process that one comes to the conclusion that the selected microcontroller is unsuitable for the application. This event would have serious consequences, as costly tools may have been invested to develop the application, not to mention the delay in the product availability with its commercial consequences.

Also, even if a microcontroller is suited to the product as it is first commercialized, this product may undergo changes during its commercial life. As a general rule, changes are always additions, never removals. If the chosen microcontroller matches current needs too closely in terms of capability, there is a risk that it could prevent the product from evolving to meet future needs. This could make the product become obsolete sooner than expected.

To summarize, it is difficult to tell in advance whether a microcontroller will fit an application. As a result, it is current practice to select a model with excess power in order to guarantee successful performance initially, and also to allow for product updates.

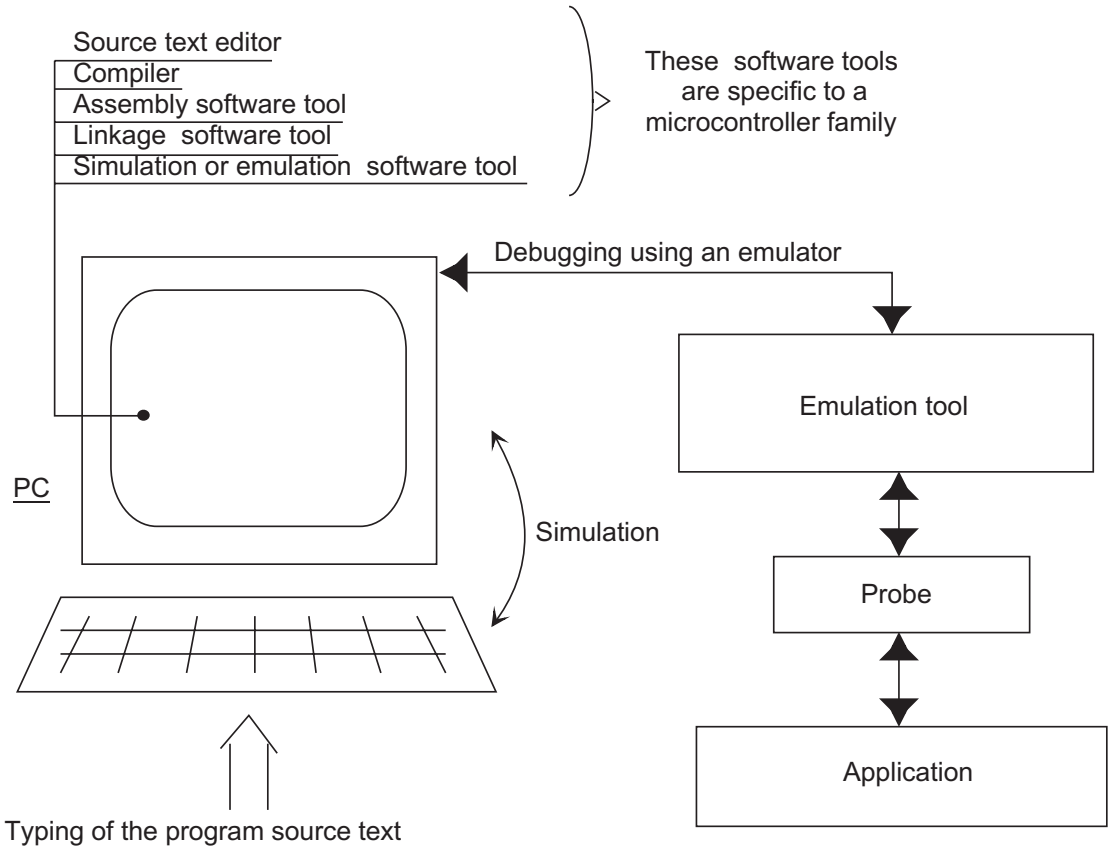
1.4.3 Choice of development tools

Once the needed power has been determined, one must investigate the development tools available for the applicable products. The first step is to compare their prices; but this is not the consideration that will determine the choice.

The real issue is how the tools will help writing the software, test it, and pinpoint its flaws. The hourly cost of a software engineer, who spends more time on software development because of the lack of efficiency of the tools, easily outweighs any savings that could have been made when investing in them.

Development tools include all that is needed to write the program, either in assembly language or in high level language, then translate it into machine language and load it into the program

memory of the application. The tools are able to test both the hardware and the software, and analyze any malfunctioning in order to allow corrections to be made. This can be done using only a Personal Computer, or external instruments connected to the computer, such as an emulator, analyzer, PROM programmer, etc. depending on the development phase. The diagram below shows where each phase takes place:



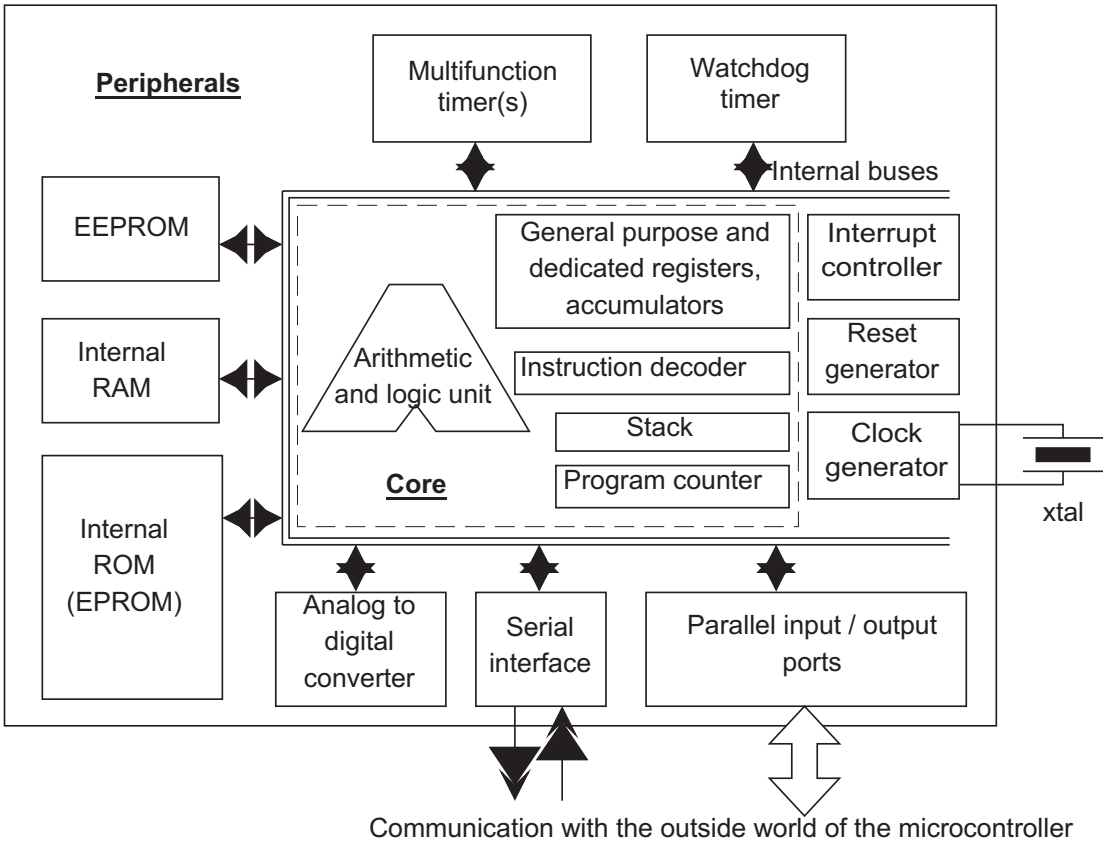
PC-based development environment

01-proc

The microcontroller itself, and the related development tools are described in Chapters 2 and 3.

2 HOW DOES A TYPICAL MICROCONTROLLER WORK?

There is a wide range of microcontrollers available on the market. They differ in their computational power, their internal organization, the number of their inputs and outputs, the type of peripherals they provide. However, a microcontroller is always a complete system on a chip that includes a core connected to some memory and surrounded by peripherals. A typical block diagram of a microcontroller is the following:



Typical block diagram of a microcontroller

02-basic

The peripherals shown here are only the most common that one can find in a microcontroller. Other peripherals, designed for special tasks or communication protocols, may be found as well. Let us mention just a few:

- I^2C serial interface
- Radio Data System decoder
- Liquid Crystal Display interface, etc.

We shall gain an overview of the main blocks in the remainder of this chapter.

2.1 THE CENTRAL PROCESSING UNIT

What is the Central Processing Unit (CPU)?

It is made up of the core, and auxiliary blocks like the clock generator, the reset circuitry, etc.

The CPU of a microcontroller is the actual programmed logic circuitry that is the heart of the application based around the microcontroller. It is where all computation and decision-making takes place. The CPU acts on data received from the outside world through the peripherals; this data is processed in a predetermined way to produce more data that will act on the the outside world.

The CPU is the part of a microcontroller that corresponds to what is usually called a microprocessor. A microprocessor contains only the computing logic; it must be surrounded with devices like memory and input-output interfaces. A microcontroller bundles all these in a single chip. For simple projects, this allows an application to be built with just one chip plus a few components. This has been made possible by progress in the scale of integration that allows powerful chips to be manufactured at a relatively low cost. This has opened up a new and very wide application field: bringing the capabilities of a computer to even the cheapest appliances. For example, nowadays home audio systems incorporate a radio receiver, a CD player, two cassette decks and an amplifier and speakers; all controlled by a common control panel with a large display that shows the FM frequency, or the CD track number and elapsed time, etc. Here, a single microcontroller performs the overall control, displays the data, responds to the keys that are pressed by the user to select the required radio channel, CD track, etc.

The word data, that is so commonly used, must be understood here in the widest sense. Though we may first think of data as numbers, data are not only numbers; they may be a wide range of objects like binary values (the state of an on/off switch), the voltage at a terminal (the wiper of a potentiometer), a character string (a piece of text), and many other things. The fact that data is thought of as numbers just comes from the fact that we are discussing machines based on binary signals. Virtually all the data processors in the world only process binary digits. These binary digits (bits) are always grouped in packs of variable lengths that are processed in parallel, thus multiplying the processor throughput by the number of these bits processed at the same time.

The first microprocessors, historically, were four-bit machines. There are still four-bit microcontrollers sold today for simple applications like telephones, washing machines, and others requiring little processing power.

In the sense that is given to this word today, a microprocessor is at least a 8-bit wide machine. The market is shared between machines of several types, with their power increasing along with the number of bits they can process in parallel. The following table gives an overview of the main classes of microprocessors today.

Table 1. Table of the main processor sizes

Data size	Relative power	Common applications
4 bits	Lowest	Watches, calculators, TV remote control, washing machines.
8 bits	Low	Industrial products and home computers in the '80s; most microcontrollers today where little numeric computation is required.
16 bits	Medium	As a microprocessor, the former PCs; as a microcontroller, industrial and automotive products used in car bodies.
32 bits	High	All PCs use this size of microprocessor today; some microcontrollers are also becoming available commercially such as automotive injection calculators.
64 bits	Highest	Only in mainframes; microcontrollers of this size are just coming out from the laboratories.

2.2 HOW THE CPU AND ITS PERIPHERALS MAKE UP A SYSTEM

The CPU cannot work alone. It is the central piece of a system that includes the following components:

2.2.1 CPU

It computes and coordinates. It controls almost all the other components of the system, except in some cases like interrupts or direct memory access where some peripherals take the initiative.

2.2.2 Memory

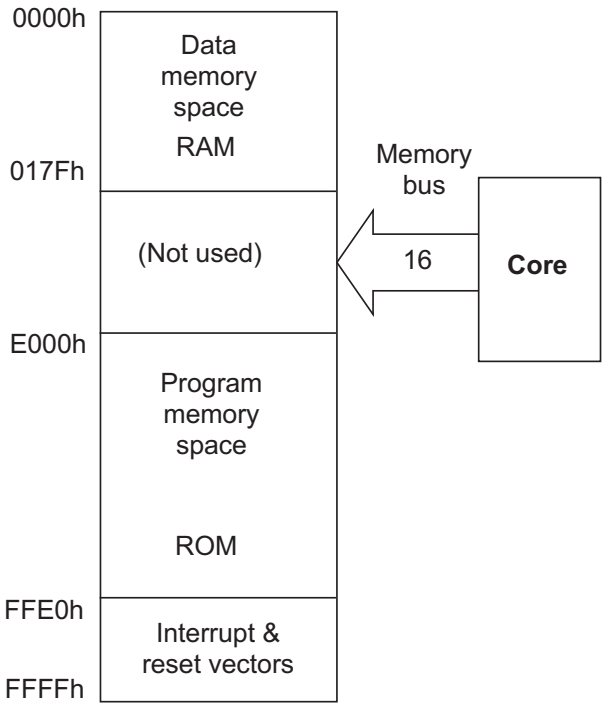
It stores both the program, that tells the CPU what to do, and the data, that is temporarily stored by the CPU like intermediate computation results, and the global state of the system.

- In computers, there is only one memory to store both. This memory is volatile, so that a supplementary, high-capacity and non-volatile storage is required to hold the contents of the memory when the system is not powered-on, in most cases a hard magnetic disk. The cost per bit stored of the memory being much higher than that of the hard disk, the capacity of the memory is usually much lower than that of the disk. Only a fraction of the disk contents resides in memory at any time.
- In microprocessor-based systems, the memory is the only storage, and various types of memory are used according to its use: read-only memory for the program, read-write memory for the data, and/or non-volatile solid-state memory for those data that must be preserved from one session to the next, the system being powered-off between two sessions.

The microcontroller has thus to handle two different kinds of things related to memory: the program, made of numbers that encodes the programming language instructions, and the data, that are what the calculations act on, and the result of these calculations. Although they are both mere numbers, they have completely different functions. Also, the characteristics of the storage are different: while the program must be kept unchanged throughout the life of the product, the data continuously change. This calls for a non-volatile, read only memory in the first case, and a read-write memory that may or may not be volatile in the second case.

The difference in roles of these two memories has led to two different approaches in the memory architecture:

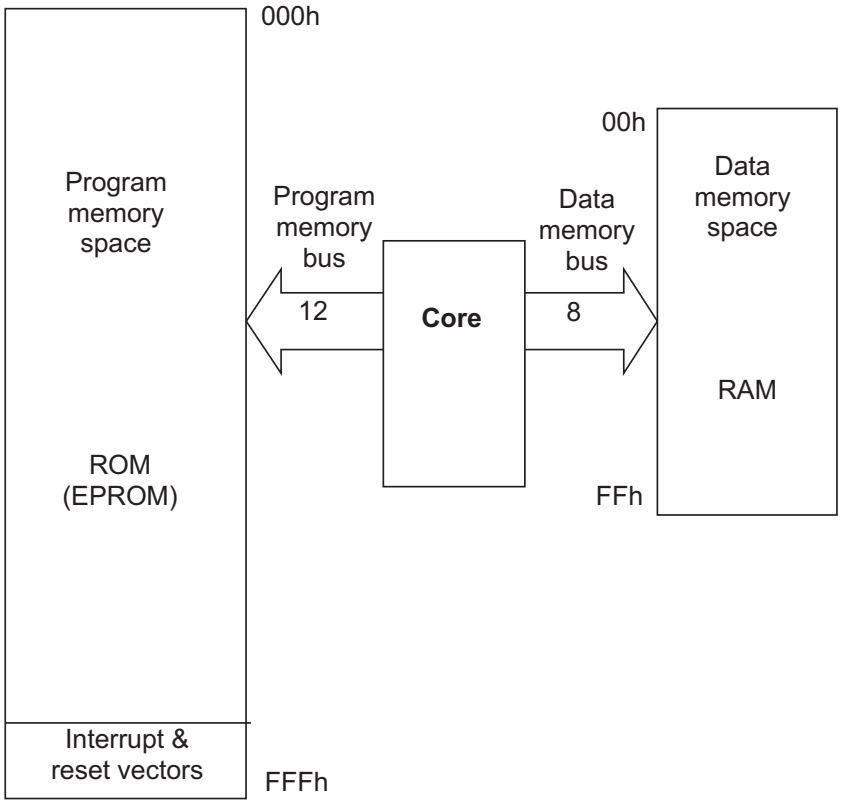
The first one, named «Von Neumann» after the name of its inventor, provides only one addressable space. The program and the data are only distinguished by the address they occupy in this space. The ST7 belongs to this category:



ST72251 memory space
a Von Neumann architecture

02-vonne

The second one, named «Harvard», provides separate addressing spaces for the program and data. No instruction can thus write anything into the program space, protecting the program from accidental changes and doubling the total addressing range. Examples of this architecture are the ST6, the ST9, and the 8051:



ST6 memory spaces :
a Harvard architecture

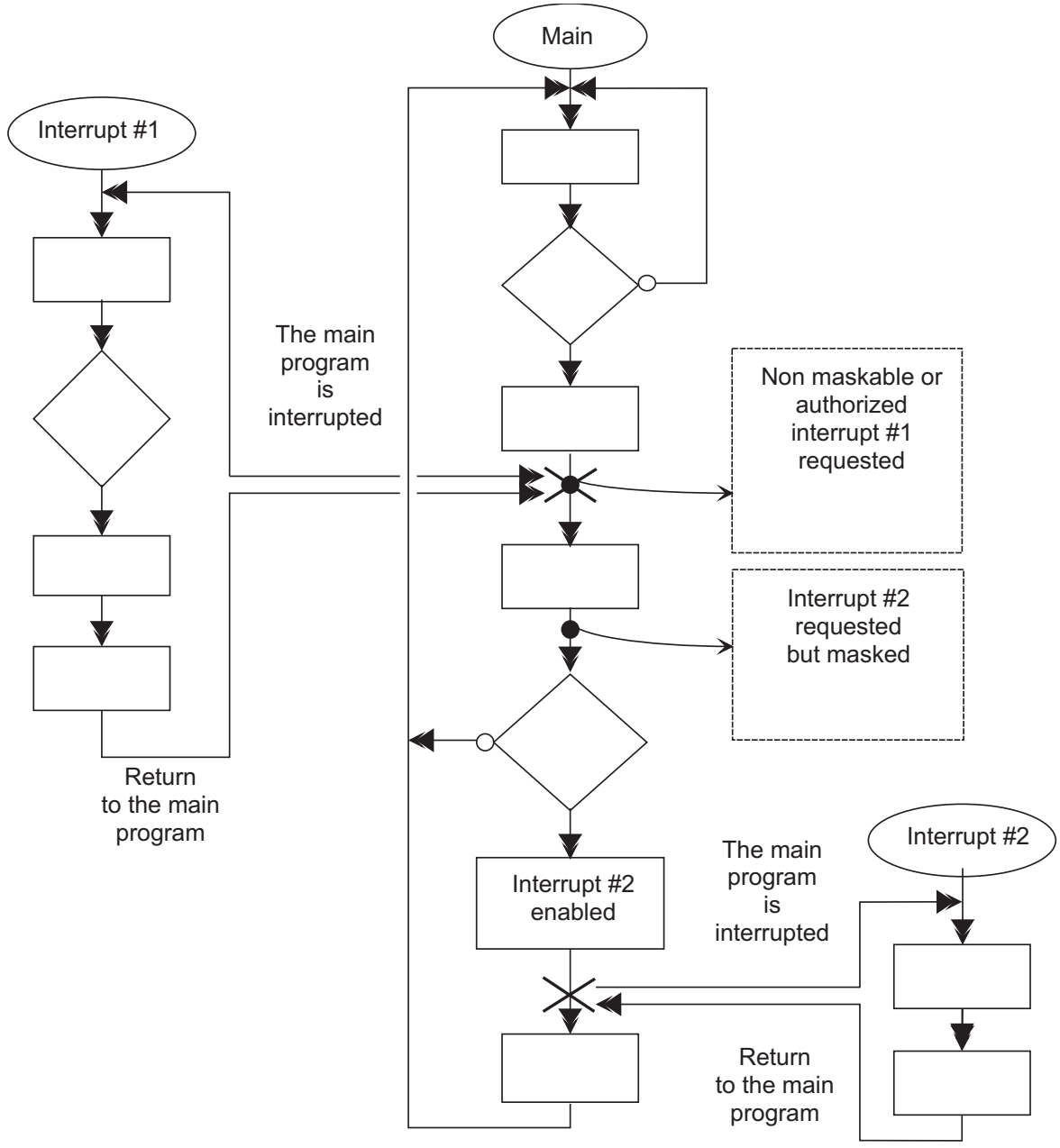
02-harvd

2.2.3 Input-Outputs

Often called peripherals, these are the point of contact between the system and the reality that surrounds it and that the system is supposed to interact with. As stated above, the data from and to the outside world are often of the analog type, and must be translated back and forth so that the system, that is fully numeric, can process them. The peripherals can be just input-output gates, for some data that are of the numeric type in the external world; or they can be somewhat complicated, if the data is either analog, or numeric but conforming to some stringent timing pattern. All the translation job performed by the peripherals saves the equivalent load to the CPU. So the total throughput of a system does not merely rely on the power of the CPU, but also on the efficiency of the peripherals.

2.2.4 Interrupt Controller

This is a piece of logic circuitry that manages the implementation of the interrupt concept described later in this chapter. Interrupts are the most common means of altering the normal course of the program, when an unexpected event (or an expected one but occurring at an unexpected time) occurs. It may be more or less complicated according to the features it provides.


Main
Interrupt #1
The main program is interrupted
Non maskable or authorized interrupt #1 requested
Interrupt #2 requested but masked
Return to the main program
Interrupt #2 enabled
The main program is interrupted
Interrupt #2
Return to the main program

Flowchart of a program with interrupt sub-routines

02-inter

Features provided may include queueing of interrupt requests, handling requests according to their priorities, or even modification of priorities to increase the chance that low-priority requests will be eventually processed in a context where there are numerous requests.

2.2.5 Bus

The bus is the set of connections that links all the components of the system and allows all the data moves, and the distribution of the address and control signals.

2.2.6 Clock Generator

This is the basic coordination circuitry that supplies a set of calibrated clock signals, at a precise frequency, that schedules all the data movement along the bus and the computations in the CPU. In some models, the clock frequency can be chosen by software.

2.2.7 Reset Generator

This circuit detects when the system has just been powered up, and resets it in a known state from which the program execution will start. The reset ensures that each time the system starts, everything occurs exactly the same way. This is a major condition for the reproducibility of the behaviour of the system.

2.3 CORE

The main components of the core are:

2.3.1 Arithmetic and Logic Unit (ALU)

This is where all the computations take place. Depending on the microcontroller used, the ALU provides a different set of operations. Roughly speaking, the basic set of operations available to all ALUs is the following:

- Addition and addition with carry, to provide for multiple precision calculations
- Subtraction and subtraction with carry
- Increment and decrement
- Bitwise shift, leftward or rightward, straight or circular (the outgoing bit in re-injected at the other end of the data word)
- Logical bitwise OR, AND and EXclusive-OR
- Logical complement

Some provide additional operations like:

- Multiplication
- Division
- and more

The ALU is connected to a register that holds the state of the last calculation done, with bits indicating (among other things) whether the result was zero, negative, or overflowed the capacity of the ALU. It thus provides a means of testing the data and changing the program flow accordingly. This register is called the status register.

2.3.2 Program Counter

This register holds the address of the next instruction to execute. It is initialized by the reset generator to a known value, called the entry point of the program. The first instruction of the program must thus be found at that address in the program memory.

2.3.3 Instruction Decoder

This circuit takes the instruction fetched from the program memory and translates their native code into its meaning, determining the actions performed by the core. The instructions fall into the following categories:

- Data processing instructions: they give the type of operation to perform (add, subtract, shift, etc.) and the address of the operand to be processed.
- Program flow control instructions: these instructions modify the value of the program counter, so that the next instruction executed will not be the one that follows the current one in program memory. They are called jump and call instructions. In particular, some of these instructions perform their action only if one or more bits of the status register have certain values, so as to jump only if, for example, the last calculation produced a zero value, or continue in sequence otherwise. These instructions provide the means of translating the branching boxes in an algorithm.

2.3.4 Stack Pointer

The stack is a storage area that has the particularity that the data put into it in a certain order, can only be retrieved in the opposite order. It is the mechanism used to handle temporary program flow disruptions, where the main flow of the program is temporarily put aside and resumed later. This is done using a pair of special instructions. The first one, named CALL, first stores the address of the next instruction to execute into the stack, before jumping to some other place. The reciprocal instruction, named RETurn, retrieves this address from the stack and jumps to the corresponding location, thus resuming the program execution.

These features give the system the capability to execute a program that reads data or binary states from external sources, performs computations, detects particular characteristics in the data, and reacts a predefined way to this before sending new data out. Using the interrupt mechanism, external events can suspend current processing and allow the incoming data to be processed and then resume the processing that was interrupted.

2.4 PERIPHERALS

The peripherals are the places where the core, that executes computer code, is in contact with the real world that is represented by electrical signals.

These signals may just be binary levels that change relatively infrequently, in which case it is easy to process them using a program. They also may change quickly, too fast for the program to handle them without imparing the computing power of the core.

In other cases, the signal is a value that belongs to a continuous range. This type of signal is called an analog value; by nature, it cannot be processed by the core, and must be converted into binary data.

An analog value may have several shapes, but it eventually falls into one of two categories:

- The data is represented by the time interval between two pulses, or by the frequency of an AC signal, or by the number of pulses of a pulse train. All these cases can appropriately be handled by a programmable timer or a UART, for example.
- The data is represented by the voltage of an input signal, or the value of a resistor that can easily be converted into a voltage. This kind of data is handled by the Analog to Digital Converter.

These considerations justify the presence of specialized peripherals, that include the required circuitry for processing the data, convert it, etc. so that it is easier to handle for the core. The less work the core has to do, the more it is available for other tasks. According to the properties of the signal, the peripheral designed to process it (we say “interface it”) may be anything from very simple to very sophisticated. We shall give here an idea of some of the most common peripherals of the ST7, starting with the simplest.

2.4.1 Parallel Input-Outputs

When the data going to or coming from the outside world is made of groups of bits, and if they can remain stable for a relatively long amount of time (at the scale of an electronic device, that may be less than one millisecond), parallel input-output ports are the right choice. They only consist of a set of gates or latches that allow for communications between the inside and the outside at times that the program chooses. This is used for example to read input switches and keyboards, and to output signals that drive lamps, motors, etc.

The capabilities of these input-outputs vary greatly from product to product. In some products, they are unidirectional or bidirectional TTL levels, fixed by hardware. In other products, they include a latch that can capture the state of the inputs on the transition of an auxiliary strobe input.

Some manufacturers, including STMicroelectronics, provide configurable input-output pins. These pins can be set as either inputs, with or without a pull-up resistor, or as an output either

push-pull or open drain. Some outputs also allow for a higher current to directly drive relays, LEDs or opto-isolators.

In addition, these pins can also be used at the same time as the input-output pins of other peripherals like Timers, Serial to Parallel Interfaces, or as inputs to the interrupt circuitry or an Analog to Digital Converter.

The configurability of these pins helps reduce the number of components in the schematic diagram, and thus the size of the circuit board.

2.4.2 Analog to Digital Converter

The ADC is a way of converting an incoming voltage into a number. The ADC is calibrated so that the relationship between the voltage and the number is well known, which allows the program to process a representative measurement of the signal.

2.4.3 Programmable Timer

This is a complex block based on a counter that can be used in many ways, so that it can either count pulses, or measure the duration of pulses or frequencies, or produce precisely timed output pulses. This peripheral is so flexible that it is virtually impossible to describe all its possible applications. In addition, the presence of a programmable timer leads the circuit designer to use it intensively, since it is the peripheral that provides the highest accuracy, when taken as a measuring device. Thus, when the measurement of a physical parameter (like a temperature, a level, a pressure, etc.) is needed, instead of designing a sensor that outputs an analog voltage, it is easier and more accurate to design it to produce a square signal with a frequency that reflects the parameter. Such signals are also easier to transport than voltages that may suffer from electromagnetic interference.

2.4.4 Serial Peripheral Interface

This interface is based on a shift register that can perform serial to parallel conversion and vice-versa. It transmits eight bits at a time, using only two or three pins. This saves pins on the the chip, and also simplifies multiplexing when connecting a number of microcontrollers together.

It can also be used to interface serial-access memory chips that provide non-volatile storage at low cost.

2.4.5 Watchdog Timer

The watchdog timer is a supplementary timer that can be used to protect the system against failures either due to the program itself (e.g. when a certain case has not been considered and the program cannot process it correctly); or a power supply brownout or electromagnetic interference has disturbed the normal working of the microcontroller. In both cases, the program may crash and the system that is built on it will no longer be stable. This can have conse-

quences in applications where the microcontroller must keep in control, like in automotive applications or in security systems.

Various solutions have been imagined to prevent such situations. The most popular is the watchdog timer. This is a timer that is set for a certain duration at power up. The program must reset it to its start value periodically; failing to do so, the timer will overflow and this event generates a hardware reset. This restores the system to the state it was at power up.

To use the watchdog timer properly, the program must reset it at the appropriate time, in a periodic manner. To do this efficiently requires some care. A word of advice is given on this subject in a later chapter.

2.5 THE INTERRUPT MECHANISM AND HOW TO USE IT

A microcontroller is a programmed computer that executes a single string of statements known as «the program». Therefore, it apparently cannot perform more than one task at a time.

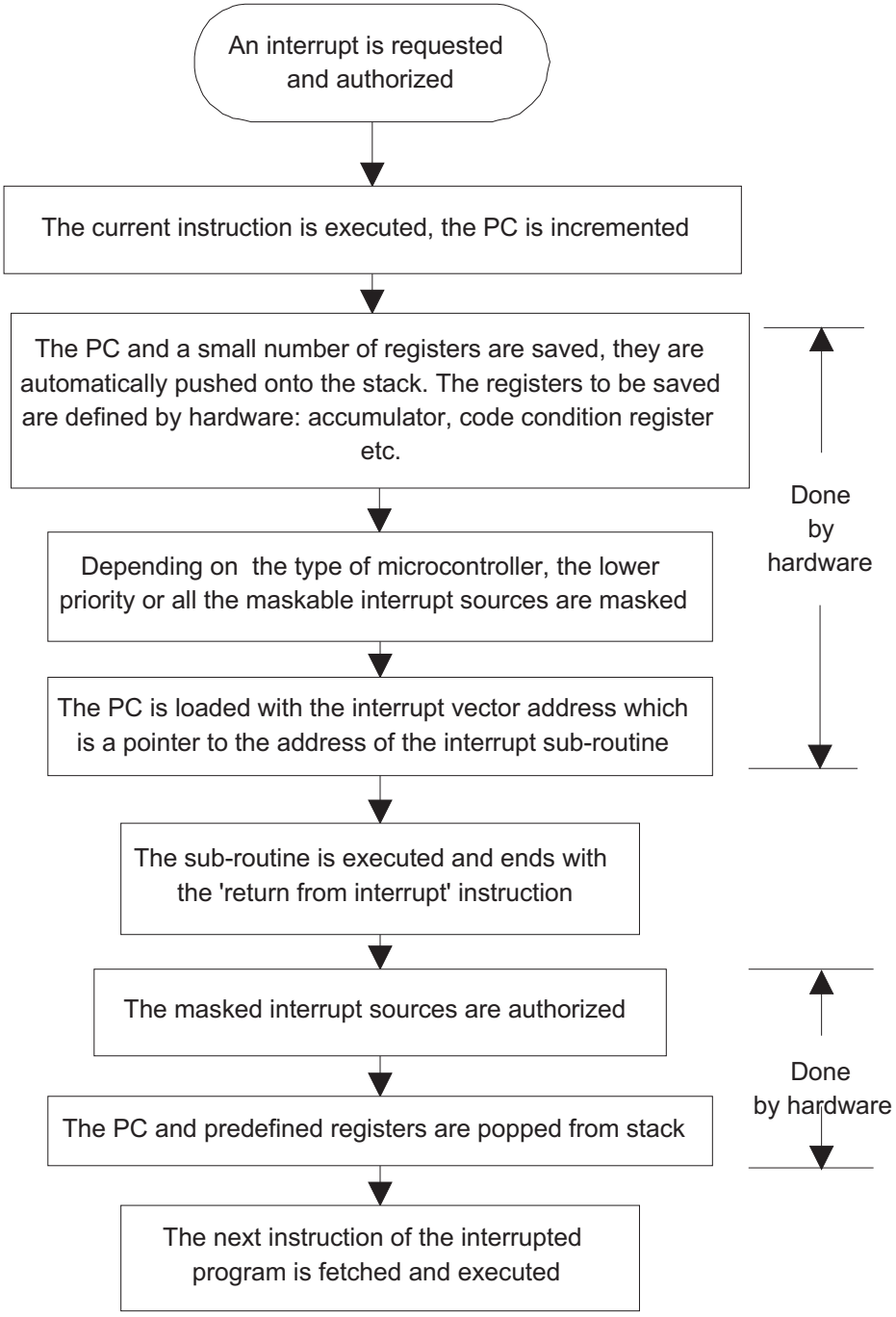
However, most if not all applications require a single microcontroller to handle many things at once. Usually, for cost-effectiveness and simplicity, the designer of a microcontroller-based system tries to pack as many functions as possible in a single chip.

The answers to this problem are based both on hardware and on software. The hardware approach is called «interrupt handling» and the software approach is called «multitasking».

2.5.1 Interrupt handling

An interrupt, in computer terminology, is a mechanism that allows the currently executing program to be interrupted, as the name implies, when an external event occurs. The computer then starts to execute a specially-written piece of code that is intended to process the incoming event. Once this processing is finished, the main program resumes exactly where it was interrupted. Nothing else happens to this program except that its execution is delayed by the time it took to process the interrupt-triggered code.

The effect of the interrupt is shown in the following diagram:


An interrupt is requested and authorized
The current instruction is executed, the PC is incremented
The PC and a small number of registers are saved, they are automatically pushed onto the stack. The registers to be saved are defined by hardware: accumulator, code condition register etc.
Done by hardware
Depending on the type of microcontroller, the lower priority or all the maskable interrupt sources are masked
The PC is loaded with the interrupt vector address which is a pointer to the address of the interrupt sub-routine
The sub-routine is executed and ends with the 'return from interrupt' instruction
The masked interrupt sources are authorized
Done by hardware
The PC and predefined registers are popped from stack
The next instruction of the interrupted program is fetched and executed

Interrupt processing flowchart

02-flow

2.5.1.1 Hardware mechanism

The hardware mechanism is important to understand. It is different for each product, so what we shall describe here pertains specifically to the ST7.

An interrupt request is a binary signal (a flag) generated by several external sources. Most peripherals of the ST7 can produce interrupt requests, for example the I/O ports, the timers, the SPI, the I^2C interface, and so on. The external cause of the interrupt request depends on the type of the peripheral: the I/O ports may have some bits configured to generate an interrupt, either on low-level, falling edge, rising edge, or both. The timer may request an interrupt on timer overflow, external capture or output comparison. The SPI may request an interrupt on end of transmission, etc.

2.5.1.2 Hardware sources of interrupt

The hardware interrupt sources are summarized in the diagram below.

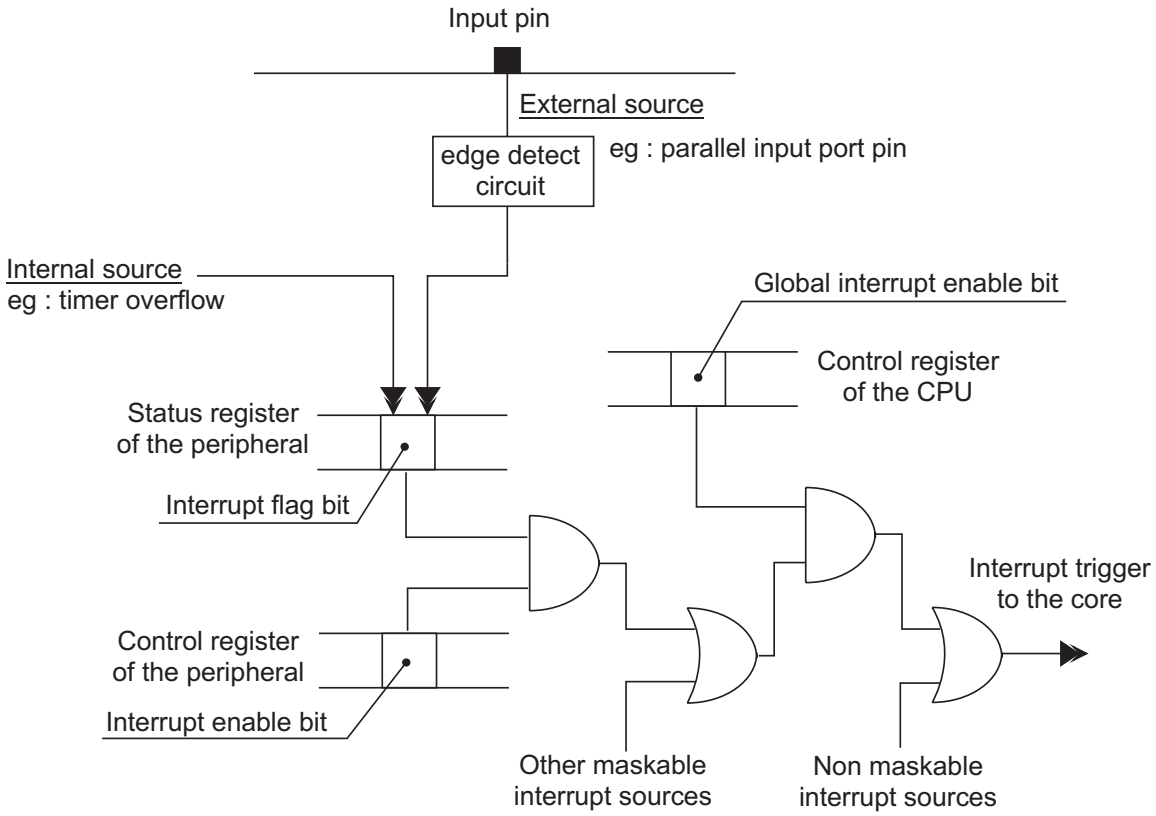


Diagram of the interrupt mechanism

02-mec

2.5.1.3 Global interrupt enable bit

The various sources of interrupt may be inhibited as a whole using the I bit in the condition code register. When this bit is set, no interrupts are generated. However, the interrupt requests are not forgotten; they will be processed as soon as the I bit is reset.

2.5.1.4 Software interrupt instruction

In addition to the hardware sources, a special instruction, TRAP, produces the same effect as an externally-generated interrupt request, but under program control. Strange as it may seem (interrupts are provided for handling unexpected events, or at least, events whose time of occurrence is not known), the TRAP instruction utilizes the whole interrupt mechanism within the regular execution of the main program.

The trap instruction triggers the interrupt processing regardless of the state of the I bit in the condition code register.

An example of the use of the TRAP instruction is the real-time debugger. When the user sets a breakpoint somewhere in the program, the debugger replaces the instruction at which the execution must stop with a TRAP instruction. The interrupt thus generated is processed by displaying on the screen the state of the microcontroller at that precise time. However, this instruction may be used in other ways as well.

2.5.1.5 Saving the state of the interrupted program

When an interrupt request triggers an interrupt, the first task of the core (after completing the current instruction), is to save its current state so it will be able to restore it after the interrupt processing is finished. This is done by pushing all the core registers on the stack. For example, in the ST7, the Program Counter, the X-register, the Accumulator and the Condition Code Register. It should be noted that the Y register is not saved, (this is because the ST7 has evolved from an architecture that did not have a Y register). If needed, the Y register should be pushed explicitly on the stack at the beginning of the interrupt service routine.

To protect the interrupt service routine from being interrupted, the I bit of the Condition Code Register is set automatically.

At this point, the interrupt service routine may execute whatever instructions the programmer chooses to write. The status of the interrupted program is known and can be restored when needed.

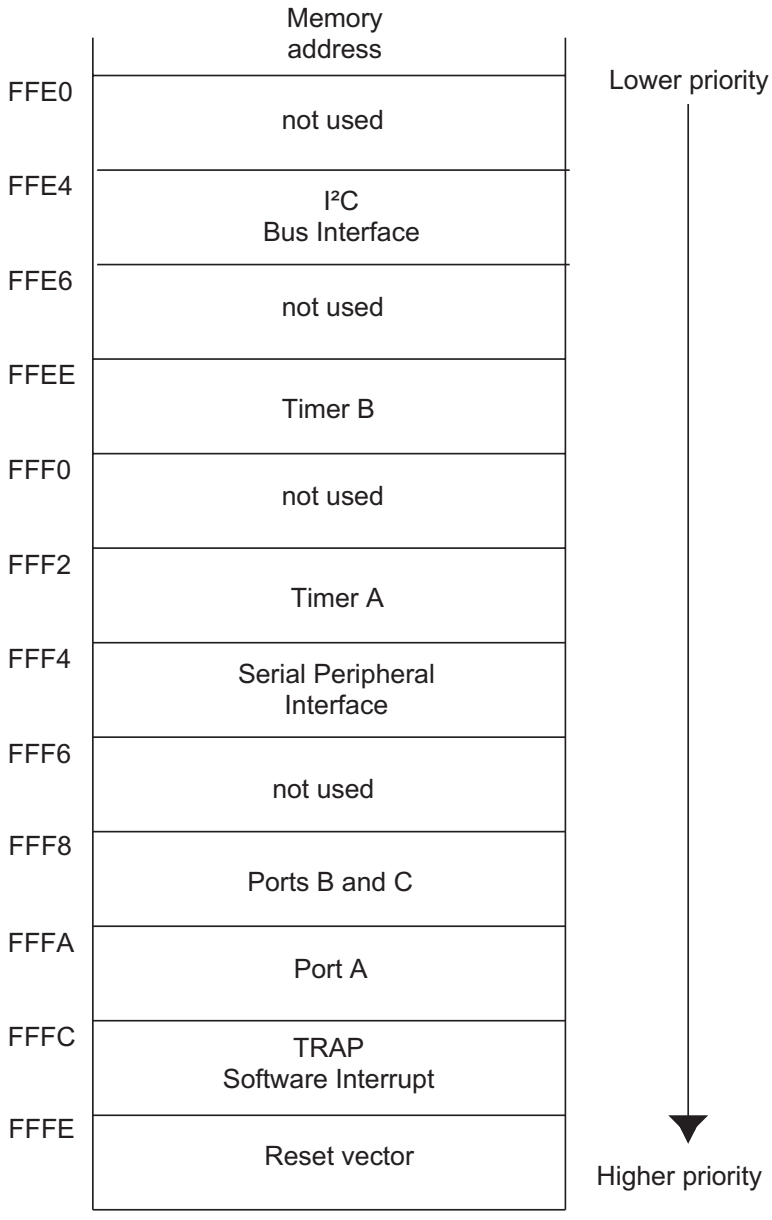
2.5.1.6 Interrupt vectorization

When the core decides to grant an interrupt request, it must know the address of the code that must be executed in such an event. This is the purpose of the interrupt vectors.

The interrupt vectors are a table of 16-bit words in program memory that contain the address of the beginning of the various interrupt service routines.

Depending on the source of the interrupt (I/O, timer, etc.), the core fetches, from a predefined location in memory, the address of the interrupt service routine especially written to process that event. The vectors are always located at the end of the addressing space. There is one vector for each interrupt source, plus one for the reset. When the microcontroller is interrupted or reset, one of these interrupt vectors is fetched in order to get the start address of either the interrupt service routine or the start of the main program.

The following table shows the interrupt vectors:



Interrupt vector table of the ST72251

02-tabv

2.5.1.7 Interrupt service routine

When the processor has granted an interrupt request, and read the interrupt vector, it starts executing the interrupt service routine. This routine is merely a segment of program, written with exactly the same ease and constraints as the main program. It may be written using the same language and tools, or in any other language.

The interrupt service routine is supposed to take appropriate action according to the source of the interrupt. For example, if an input bit has changed its state, the service routine may change the state of an output bit; if the interrupt was generated by the timer, this may produce the transmission of a byte by the SPI, etc. according to the structure of the application as defined by the programmer.

Eventually, the service routine is finished. Then the core may return to the main program. This is done by executing the IRET instruction.

2.5.1.8 Interrupt Return instruction

As described above, an interrupt service routine looks a little bit like a subroutine. Like in a subroutine, the return address is stored in the stack, and the execution of the RET instruction returns to the calling program.

However, some more things have to be done before returning to the interrupted program. All the core registers were pushed on the stack when the the interrupt request was granted (except the Y register). They must now be restored, so that the execution of the service routine will not leave any trace in the core. This is the role of the IRET instruction in the ST7.

The IRET instruction proceeds by popping all the data off the stack that had previously been pushed, namely the Condition Code register (at this point the I bit is also restored), the Accumulator, the X register and the Program Counter.

From this time on, execution of the interrupted program resumes.

2.5.2 Software precautions related to interrupt service routines

As described above, the interrupt mechanism is fairly simple to use, since it only consists of setting the interrupt vectors to the address of the corresponding service routines, and writing a piece of code that must end with a IRET instruction.

Actually, an interrupt service routine may do anything in the system, since it uses the regular instruction set of the core and has access to the whole memory. It may thus affect the state of the main program, even though the core registers have been preserved. The following paragraphs deal with the precautions to take when using interrupts.

2.5.2.1 Saving the Y register

(This point is specific to the ST7.) If the service routine uses the Y register, one must remember that this register is not saved automatically by the interrupt granting mechanism.

Thus it is up to the programmer to save it, by pushing it to the stack at the beginning of the service routine, and popping it before executing the IRET statement.

2.5.2.2 Managing the stack

This is just a reminder, since it applies anywhere in the program. The service routine must track the usage it makes of the stack, so as to pop at the end as many bytes as it had pushed at the beginning. This may look trivial, but if some pushes occur in some conditions and not others (i.e. the service routine has conditional statements somewhere), the popping must occur in exactly the reverse way, taking into account the same conditions as those that produced the pushing. This may not be very obvious to code.

2.5.2.3 Resetting the hardware interrupt request flags

In some peripherals, the hardware flag that produced the interrupt request is automatically cleared on servicing the interrupt. In this case, no special care need be taken.

On the contrary, in some other peripherals (such as the timer), the interrupt request flag keeps its state after the interrupt is granted. This flag must be cleared anywhere in the interrupt service routine, but necessarily before the I bit of the Condition Code Register is cleared (on execution of the IRET instruction). Otherwise, the interrupt service routine would be called again immediately after executing the IRET instruction, and the core would loop indefinitely through this interrupt service routine, thus blocking the main program.

How to reset the interrupt request flag is described as part of the description of each peripheral.

2.5.2.4 Making an interrupt service routine interruptible

On interrupt granting, the I bit of the Condition Code Register is set, to prevent the service routine being interrupted by incoming interrupt requests. Further interrupt requests will then suffer from a delay before they are serviced. This delay is called «interrupt latency». Actually this term includes the reaction time of the core itself, to which the time for the servicing in progress must be added.

However, there are cases where it is necessary to allow an interrupt service routine to be itself interrupted. This is the case if a service routine performs processing that takes a certain amount of time, and another interrupt source requires that its request be processed immediately, i.e. the permitted latency is short.

The solution is then to allow the slow service routine to be interrupted. This may be done by resetting the I bit of the Condition Code Register. This must be done after the hardware interrupt request flag that triggered the interrupt currently in progress is cleared, for the same reason as explained above. Please note, however, that this must only be done when necessary, since the size of the stack is often limited in small microcontrollers.

2.5.2.5 Data desynchronization and atomicity

This paragraph addresses the precautions that must be taken, in the main program or any service routine that may be interrupted.

In many cases, data is organized in blocks in memory, that is, several bytes, successive or not, make up a piece of data. Some coherence rules must be followed when using these data. Failure to observe these rules may produce unexpected results and, very likely, an application crash.

When no interrupts are used, the main program can easily follow these rules by taking care to perform all data changes in the appropriate order and respecting the predefined relationships between each of the bytes that constitute a piece of data.

When interrupts are used, respecting the coherence rules may become more complex. Actually, an interrupt is an asynchronous action that can occur at any time. Let us assume that the main program is currently altering one piece of data that is made of several bytes. It first writes some bytes, then more bytes until it is finished with a new data in memory.

If an interrupt occurs in the middle of the process of altering the data, the following risk may appear. If the interrupt service routine uses the data that the main program is writing, the service routine may get data that fail to follow the coherence rules, since not all bytes have been updated yet. The service routine may then be misled by an incorrect value that it cannot handle properly, or just interpret that data differently from what it was expected to mean if it had been fully modified. This may have very serious consequences on the working of the application. This circumstance is called «data desynchronization».

To avoid this, some precautions may be taken in cases where interrupt service routine may find incoherent data. They all ensure that all the data will be updated at once, and never used unless completely updated. This condition is called «atomicity», from a Greek root meaning «that cannot be cut». To properly handle multi-byte variables that are shared by a main program and an interrupt service routine, or by two interrupt service routines of which one may interrupt the other, the handling must be made “atomic”.

The following example shows:

- What happens when the data are desynchronized.

Let us assume the main program wants to increment the word variable reg (16-bit register in page zero), that currently contains 42FF hex. The following code will be used:

```
; word variable reg contains 42FF hex
; increment word variable reg by 1

            inc reg+1                   ; add 1 to the low byte
                                        ; low byte is incremented. reg
                                        = 4200 hex

            jrne endinc                 ; increment high byte if carry

endinc : ...                            ; here the program continues
; both bytes are incremented. reg = 4300 hex
```

If an interrupt service routines uses the value of reg, and if the interrupt request occurs for example at the first line of the code above, the interrupt service routine will see that reg is 4200 while it is actually either 42FF or 4300. This error may have serious consequences.

- How to make the handling atomic.

To avoid this situation, it is sufficient to mask out all the interrupts, by changing the code as follows:

```
; increment word variable X by 1
            sim                         ; prevent interrupts from
                                        occurring
            inc reg+1                   ; add 1 to the low byte
            jrne endinc                 ; increment high byte if carry

endinc      rim                         ; allow interrupts to occur
            ...                         ; here the program continues

; both bytes are incremented. reg = 4300 hex, now the interrupt can be
                                        performed
```

All interrupt requests that occur between the SIM and the RIM instructions are delayed for the duration of that piece of code. If this would cause an excessive latency for one particular interrupt that does not use that data, it is possible to mask out the specific interrupt source whose service routine actually uses this value.

This example mentions the case where the data is written by the main program, and read by the interrupt service routine. Actually, the reverse case is also a source of problem: if the main program reads the data, and the interrupt service routine writes it, the main program may start

reading the first bytes of the data, then the interrupt occurs; on return, the remainder of the data are read, but unfortunately there may not be coherence between the first byte that was read before the interrupt and those read after it.

2.5.3 Conclusion: the benefits of interrupts

The interrupt system is a very appropriate means of processing events that have the following features:

- They are triggered by a hardware signal coming from outside. Theses signals are connected to appropriate pins of the microcontroller; or they are the result of the working of internal peripherals that reach a certain condition, for example the internal timer has overflowed. Though the timer is built-in the same chip as the core, it is functionally considered external to the core.
- They occur at their own time, and thus unexpectedly for the main program.
- They require a quick reaction from the core, either because they occur frequently or because the status of the external device that requests the interrupt would not keep its meaning after too long a delay.
- They do not require complex processing; typically, they require reading some data from outside and storing it to memory, or transferring data from the memory to the external circuitry.

2.6 AN APPLICATION USING INTERRUPTS: A MULTITASKING KERNEL

The conclusion in the previous paragraph states that interrupts are well-suited for a certain class of events to be processed. However, there are other cases outside this category. Some of them are better addressed by the concept of multitasking.

In many applications, several processings are required that do not match the specificity of the interrupt-driven processes. For example:

- Two or more processes are continuously active, that each take long processing times.
- These processes are not (or not directly) started by external events.
- They do not require a quick reaction time.

In such cases, the interrupt concept is obviously inappropriate. Actually, these processes seem to require each a core of their own. However, considerations of cost may not allow for multiple microcontrollers on the same board. The concept of multitasking is the answer to this requirement. It is a software solution that does not require extra components, and makes the system believe the various tasks run on different cores, although it is simply the same core that is shared between all the tasks at the expense of the computing power that is shared between this tasks, plus a certain waste produced by the specific mechanisms that provide for the multitasking. This waste limits the frequency at which the tasks can be switched; if they are switched too often, the proportion of the time taken to switch tasks becomes too large, and the corresponding part of the microcontroller computing power is lost. The designer must check whether the power remaining for each task is sufficient or not; if not, the type of microcontroller is probably unsuitable for the project.

There are two kinds of multitasking, namely pre-emptive multitasking and non pre-emptive multitasking. The second kind is also called cooperative multitasking.

2.6.1 Pre-emptive multitasking

Pre-emptive means that the computing power that is allocated to a task is withdrawn from it without notice, that is, that particular task is stopped at an unexpected place by brute force. Then, the power is allocated to another task, until it is stopped in turn, and so on for all the tasks; then, the first task that is currently sleeping regains control and continues for some time.

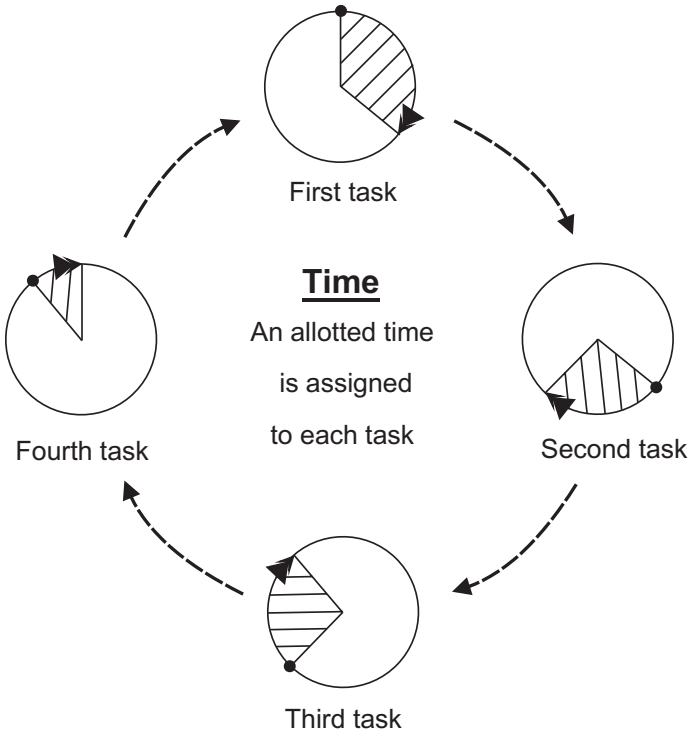
The task switching is done under control of an interrupt triggered by a timer. This allows the core time to be partitioned at will between the various tasks. The time for which each task is allowed to run may be the same for all tasks; or it may be decided to allow more time for some more important (or time-consuming) tasks, and less for others. In a word, the multitasking kernel may fine-tune the resource sharing between the tasks.

The main drawback of this system is that since the tasks are interrupted at any place in the code, many precautions must be taken to ensure the coherence of the data, just as explained above about interrupts. In fact, if a task starts to write a piece of data and is put asleep in the

process of updating the data, and another task uses that data, there is a risk of desynchronization. The same type of precautions must be taken to ensure atomicity of data updates. The same problem may also occur if more than one task handles control sequences for an external device. If this external device needs a precise control sequence that must be completed before a new sequence is started, there is a risk that a task may lose control before the sequence is complete and control may be transferred to a task that attempts to use the same device. The attempt may then fail or interfere with the unfinished sequence of the previous task. Here again, a protection mechanism is required.

In summary, the advantage of pre-emptive multitasking is that task switching is done automatically and independently from the code of each task; the relative power attributed to each task may be adjusted to fit the requirements of each task.

The drawback is the opposite of the advantage: since the task switching happens at any time and any place in the code, the programmer must locate the critical areas of code where special protection mechanisms must be included. This may be more difficult than it might appear, for it is not always easy to find all the possible collisions and keep them from happening.



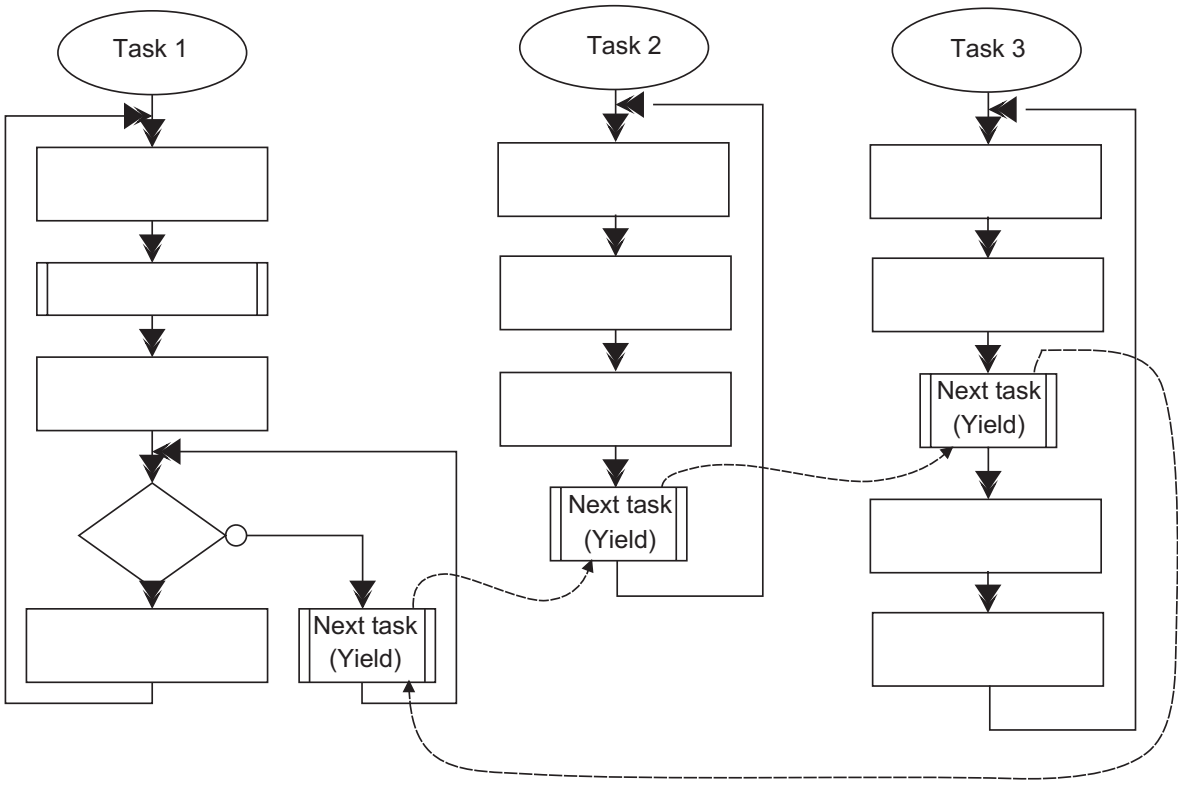
02-preem

2.6.2 Cooperative multitasking

Cooperative multitasking draws its name from the fact that task switching is not spontaneous. It only occurs when the task-switching function is called by the currently active task. This implies two facts:

- Task switching occurs only when the code decides it. As a consequence, it is easy to avoid data desynchronization and access collisions by placing the task switching calls at the proper places.
- The partitioning of the core time between tasks cannot be set at will, since it is not possible either to insert into the code as many calls to the switching function as necessary or to put them at the right places to control the time intervals allocated to that task.

The reader can easily see that the strong and the weak points of one type of multitasking are the opposite to those of the other system. This explains why both are used, the type being chosen to best match the application's requirements. In addition to theses features it is fair to say that cooperative multitasking is easier to implement and less resource-consuming than its competitor.



Cooperative multitasking ; simplified example

02-coop

2.6.3 Multitasking kernels

2.6.3.1 Advantages of programming with a multitasking kernel

The multitasking kernel is the piece of code that controls the multitasking. The way it does it, and the flexibility it offers may vary greatly from one kernel to the other. It is generally supplied ready made, and the programmer has to give his program the appropriate architecture to get the benefits of it.

Writing an application with multitasking in mind is easy and leads to a clear, organized structure. The work to do is divided in tasks, and these tasks are written separately as procedures. They exchange data using either communication mechanisms built-in to the kernel, or common data in memory, taking care to avoid collisions. The main difficulty is to identify what a task actually is. For example, two processings that are always performed one after the other, and always in the same order, constitute a single task. On the contrary, two processings that either may or must be performed at the same time are two separate tasks.

The features of a kernel vary from product to product. One must first know which kernel is being used, the type of multitasking (pre-emptive or cooperative), and the services provided by the kernel. Some of these are discussed below.

2.6.3.2 The task declaration and allocation

The kernel must be aware of the existence of the tasks, their number and their start addresses. Some kernels expect this to be stated at compile time; in this case, the number of tasks is known from the beginning of program execution and cannot change afterwards.

Some others allow tasks to be added (or created) while the program is running. This allows tasks to be created when the need shows up, for example to process an incoming event and then terminate. In this case, it is convenient to also have the ability to remove or kill the task. Again, there are two options: a task may terminate when it decides to do so (kill itself or “suicide”); or it may be killed by any task, including itself.

2.6.3.3 Task sleeping and waking-up

A task may be alive but have nothing to do; in this case, to save computing power, it is wise to completely stop allocating core time to it. The task is then asleep. This can be done by calling a function often called Sleep, passing to it the identification of the task to be put asleep. A task can put itself asleep if it is waiting for some event.

Obviously, it is necessary to have a means of waking-up a sleeping task. This cannot of course be done by the sleeping task itself, and can only be done either by other tasks or by an interrupt service routine. For example, let us consider a task that processes the keystrokes generated by a keypad. The hardware of the keypad may generate an interrupt when a key is pressed. This interrupt may wake-up the keypad task that reads the keycode and takes the appropriate action. When this is done, the task may go asleep again. One might ask why it is

not simpler to perform the processing right in the interrupt service routine, instead of this apparent complexity. Actually, the processing of the keystroke can take a long time, too long to allow it to freeze the remainder of the application as happens when an interrupt is being serviced.

Other services may be supplied by the multitasking kernel, coping with priorities, intertask communication, etc.

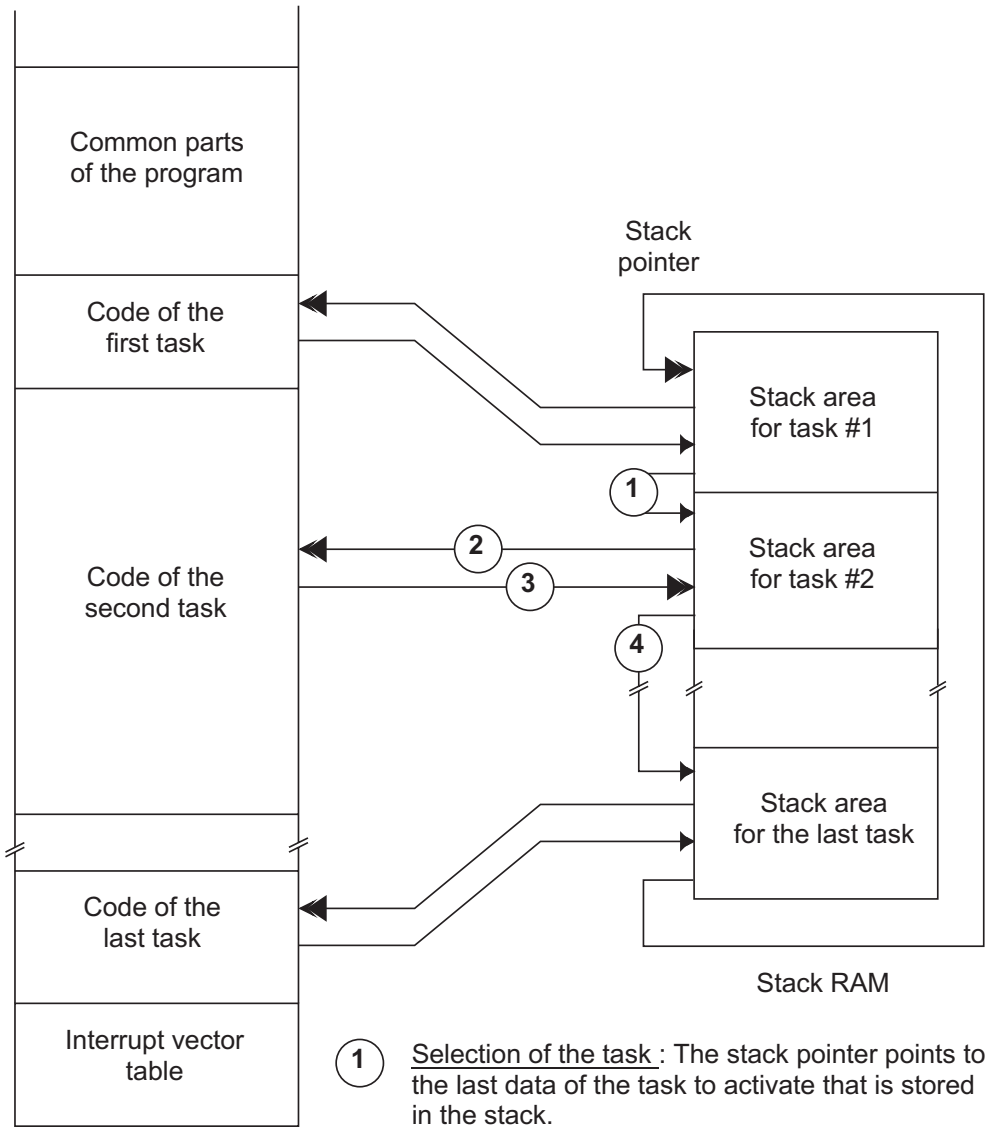
2.6.3.4 Multitasking kernel overhead

The purpose of a multitasking kernel is to share the power of a single processor or core between several tasks. Obviously this sharing means less power for each task, except perhaps if all tasks but one are asleep. However, even in that case, the task does not benefit from the whole computing power, since some of the core power is drawn off by the kernel itself. In addition, it is equally obvious that the kernel is a piece of code that occupies a certain amount of program memory as well as data memory. But overall, the main concern with memory requirements relates to the stack.

The stack is the place where the state of a program is continuously stored. This amount of data is often referred to as the “context”. Each task has its own context, which consists of all the return addresses of all nested procedures and functions; most compilers also store the function arguments and local data in the stack. This can add up to a large amount of memory, not to mention that some free space must remain in the stack to handle interrupts that may themselves consume a certain amount of stack, in the same way as the main program.

This implies generally that the stack space must be very large, since the amount of data mentioned above must be multiplied by the number of tasks alive at the same time.

The ST7, being a small 8-bit core, provides for at most 256 bytes of stack. This allows for a multitasking kernel with a limited number of tasks and services.



1 Selection of the task : The stack pointer points to the last data of the task to activate that is stored in the stack.

2 Activation of the task : The popping of the data off the stack restores the tasks as it was before being unselected.

3 Deselection of the current task : The state of the program is saved in the stack before it is deselected.

4 Selection of the following task : The stack pointer points to the last data saved from the next tasks to activate.

Working mechanism of a simplified multitasking kernel

02-task

3 PROGRAMMING A MICROCONTROLLER

A microcontroller is basically a programmable component. This means that it can do almost anything, when properly programmed.

In fact, the design of the electrical schematic of a microcontroller-based application raises few questions; the input and output pins of the microcontroller are simply connected to the binary signals either produced or used by the external application. The designer only has to take care to select the right pins, since some signals must be connected to special peripherals like the Analog to Digital Converter, the Timer, etc.

It is the program that configures the pins so that they have the correct electrical behaviour, and that processes the data to produce the appropriate response to the input signals. Since the board will be designed with as little electronic processing as possible, all the processing is done by software.

This produces the flexibility that is the main feature of any programmed system: unless a hardware problem arises, most of the fixes and changes done to a programmed system will be done in software.

Programming the processor is thus the key activity of the designer, the one that will take the largest part of his time. For this reason, the use of the right tool to program the application is critical, since program design and testing time can vary greatly according to the tools and the language chosen.

This chapter addresses the two main issues the programmer faces: selecting the appropriate language for the best productivity, then selecting the appropriate software tools that will allow not only to program in that language, but also to test the program written in that language. We will learn that an investment made prior to starting the design can prove very efficient in terms of development time, and therefore, pay for itself.

3.1 ASSEMBLY LANGUAGE

3.1.1 When to use assembly language

Assembly language is the native language of each microprocessor. It used to be the only way of programming a small microcontroller until high-level language compilers were made available. Programming in assembler was a job that required a lot of care and very many lines of source code relative to the size of the application. It was justified when program memory was small and assembly language was the only way to optimize the code size. Nowadays, microcontrollers, except for those at the lowest-end, can afford enough memory to cope with the code expansion factor inherent in high-level languages. Thus, for reasons explained in the following paragraph, a high-level language is strongly recommended and using assembly language should not be considered except when absolutely needed.

There are, in almost all applications, parts of the code that still require assembly programming. These parts are, most of the time, small but have an important impact on the program. Here are a few such cases:

- The initialization part of the program. All high-level languages provide for initialization of the core and the memory. However, the basic organisation of the memory (address of the ROM and the RAM, reset and interrupt vectors) and a few other kinds of initialization are supplied as an assembly-language template, that has to be adapted to suit the actual application.
- Some interrupt service routines that require very fast processing.
- Some repetitive functions that are frequently invoked and whose optimization in terms of speed has an important impact on the performance of the whole program.

The third case implies that the programmer carefully reads the implementation chapter of his compiler's manual. The way arguments and return values are passed back and forth are specific to each compiler. Failure to comply with these conventions will prevent the assembly code from working.

3.1.2 Development process in assembly language

The development of a program consists of three main phases: analyzing, writing the code, debugging. In other words, the successive phases can be described as follows:

- The first phase is when the programmer defines what the program should do. This is only paperwork, even if it is done using a computer and a word processor or a spreadsheet. This phase defines the main program blocks, the data inputs and outputs, the storage, and some of the algorithms.
- The second phase is the translation of the first one into the chosen computer language. The result of it is the source code and a few files that drive the various programming tools. The tools used in this phase are the text editor, to type and amend the source code, and the assembler and the linker to check its syntactic correctness. The Make utility is also a convenient tool, that helps keeping the program up-to-date, when any of its parts have been changed, by processing only the changed source files.
- The third phase consists of all that is needed to make the source code work. It involves removing all programming errors, that is, the flaws in the first and second phases related to logic, coordination, and data management. This third phase is by far the most difficult and requires the most development time. The tools used, the Simulator and In-Circuit Emulator, have to be very powerful because many errors are difficult to find.

When the program is fully functional, it is often stored in an EPROM that is either external, or as in the ST7, internal to the microcontroller. This produces a prototype of the microcontroller that must be extensively tested before being launched to production, especially if, for large

quantity production, the microcontroller includes a masked ROM programmed by the device manufacturer and that cannot be altered afterwards.

3.1.2.1 Assembly language

Assembly language is merely a set of mnemonics that duplicates each instruction in a more legible way, to help writing the programs. Machine language is just a series of numbers that obeys a certain code to indicate to the core which instruction to execute and which data to use. Assembly language provides acronyms that are easier to remember. The following example shows an excerpt from an assembly listing. The first two columns show numeric data, which represent the addresses of the instructions and the machine-language code. The Source line column shows the machine code in mnemonic language. It is obvious that, provided one has learned that `ld` means the instruction Load, and that, of the two operands of this instruction, the destination operand comes first, it is easy to understand the first lines of source code as documented in the comment column:

```
Loc    Obj. code   Source line                Comment
------ ---------   -----------             -------------
000000 A6FF             ld A, #$FF   ; load accumulator with immediate
                                        data FF (hex)
000002 AE64              ld X, #100   ; load X register with value 100
000004 F7          loop: ld (X), A    ; load location pointed to by X
                                        with contents of A
000005 5A                dec X        ; decrement X register
000006 26FC             jrne loop    ; jump to label loop if X not equal
                                        to zero
```

This short program is a loop that fills memory addresses 100 to 1 (decimal) inclusive with the value FF hex.

Programming in assembly language involves using a text editor to write a text file that obeys the syntax and conventions of the specific assembler that is to be used. It should be noted that assembly language is not standardized in any way, for two reasons:

- The instruction set changes from one processor to another
- For a specific processor or microcontroller, software tool sets from different suppliers each have their own syntax, although two source files written for two different tool sets may seem very close.

Thus, to write an assembly language source file, the programmer must first know which processor or microcontroller will be chosen for his project, then which software tool set he will use. Then, he has to learn both the characteristics of the processor's instruction set (instruction types, addressing modes, etc.) and the specific syntax of the assembler he will use.

Then, once he masters both, he may start to write his source file. Obviously, if later he has to do the same job with a different processor, the source file will be of no use in the future project.

3.1.2.2 Assembler

The word assembler has usually two meanings. Properly used, it is a program that translates a text file, written according certain rules, into a binary file that contains a series of instructions specific to a microprocessor or a microcontroller. However, the word Assembler is often improperly used instead of “assembly language”, for example in the sentence: “this program is written in assembler”. This paragraph introduces the translating tool, or “assembler”.

The assembler is a program that runs on whatever computer is used by the programmer for his regular job. It takes the source file, as explained above, as an input; then, after translation, it outputs, depending on the user-specifed options, any of the following files:

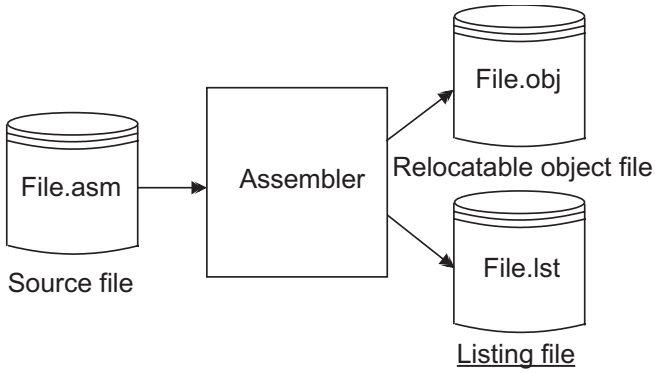
- The object file, containing binary data intended for further processing. It can not be read by man.
- The listing file, is a report containing both the original source code and its numeric translation, presented in a tabulated manner. It can be read by man and used for reference purposes.
- In some cases, other files like lists of variables and labels, or additional data. This varies from one assembler to another.

If the assembler encounters an error in the syntax of the source file, or some ambiguity or lack of information preventing it from completely processing the source file, it generates an error report, this may be output in the listing, in a separate file, or directly on the computer console.

The error messages tell where the error is located, and, as much as possible, the cause of the error, as in the following example:

```
TRIAL.ASM(6): ERROR: not a ST7 instruction or directive
```

Here again, the format of all the above mentioned files is chosen by the assembler's manufacturer, and differs from one to another.



Assembler invocation

03-asm

3.1.2.3 Linker

If the whole application program is small, it is easy to put the source text into a single file. The assembler then produces an object file that contains the whole machine code ready for use. This is called absolute assembling.

However, there are very few applications so simple that their source text occupies only a few pages. In most cases, the total source text amounts to thousands of lines that may represent hundreds of pages. In such a case, it would be impractical to edit a single large source file. Not only would the assembly process take a long time but this time would be spent whenever a change was made to the text.

It would be better to divide the whole text into several files, and to assemble them separately. This way, a change would affect only one of the files, which can be quickly re-assembled.

Working this way requires additional features:

- A means of telling that a particular source file references a data variable or a label that is defined in another source file, in order to prevent the assembler from merely stopping and issuing an error message saying that something referred to has not been found defined in the same source file.
- A tool to glue all the generated object files into a single object file, ready for use.

The first issue is addressed by additional syntax features that typically provide declarative statements such as “External” and “Public”. The “External” statement declares that a certain label referenced in the source file will intentionally not be found there, since it comes from another source file. The “Public” statement declares that a label (or a variable) defined in this file will be referenced by another source file.

The tool that glues together all the object files, each the result of assembling the source files, is called the Linker.

The linker performs the following tasks:

- It takes all the object files, and merges them into a single object file by concatenating them one after the other.
- It corrects the address values in all the instruction operands that refer to objects whose location in memory has been set or altered by the concatenation.

To do this, in addition to the set of object files, the linker requires a control file that tells it the list of the object files to link together, the order in which they must be put, and the absolute addresses at which the result will be installed in the microcontroller's memory.

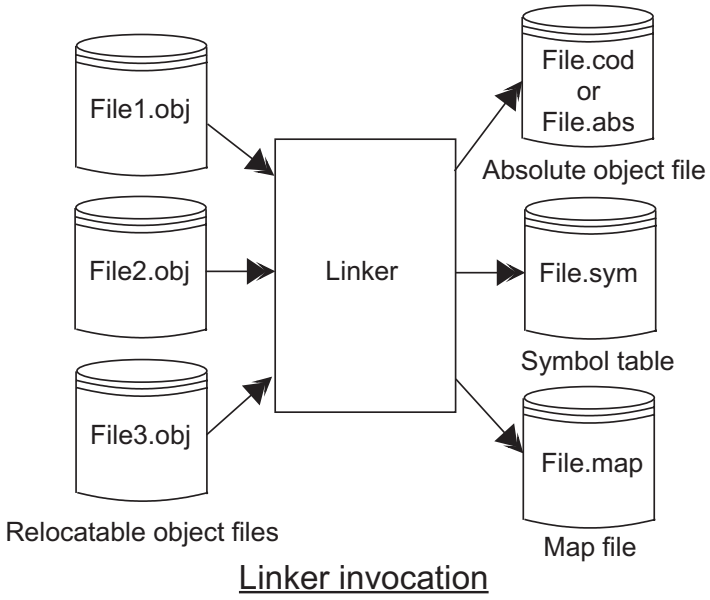
This may look complex; but actually, it make processing large programs easier and faster. When a change has to be made to a particular source file, only this file must be edited. Then only this one must be re-assembled, which saves time; then, all the object files must be linked again. The linking process is fast, compared to assembly, since the files contain binary data and their format is optimized to make the linker's job easier.

If a hardware change is made to the board that changes the address of some memory or a peripheral, it may be only necessary to alter the linker control file and link the program again.

Once the program is linked, the resulting object file contains the whole program. This file is said to be an absolute object file, meaning that all addresses are defined; as distinct from object files before the link process that are said to be relocatable, i.e. their addresses can be changed later.

The absolute object file can be used either to be downloaded into an emulator to check and debug the program; or into an EPROM programmer to program the chip that will hold the code.

As for the assembler, the linker is built to process relocatable files produced by the assembler frim the same tool set. It is usually not possible to mix the tools, like assembling with the assembler from supplier A then linking with the linker from supplier B.



03-link

3.1.2.4 The project builder/make utility

In the process introduced above, that relies on splitting-up the processing to save time, it is important to keep track of which sources have been altered, only re-assemble these and not the others and then link all the object files. Forgetting to do this may waste a lot of time. For example, after making a correction, if a program still shows the same incorrect behavior as previously, one is tempted to suspect another part of the code. Trying to identify which other part of the program produced the problem will be a useless effort, if the real reason was that the file containing the change had not been re-assembled and linked.

Thus, a tool that can guarantee that the programmer will never forget to reprocess his files is an invaluable help. This tool is called Maker or Make utility.

The maker is a tool that works under the control of a file that gives the names of the source files, the name of the output object file, the names of the tools needed to process a file to another file (assembler, linker), and the dependency relationships between the files.

For example, the dependency relationships specify that the final object file is produced by applying the linker tool to the files named in the linker control file; that each of the individual object files is produced by applying the assembler tool to the corresponding source file, etc.

The maker works as follows:

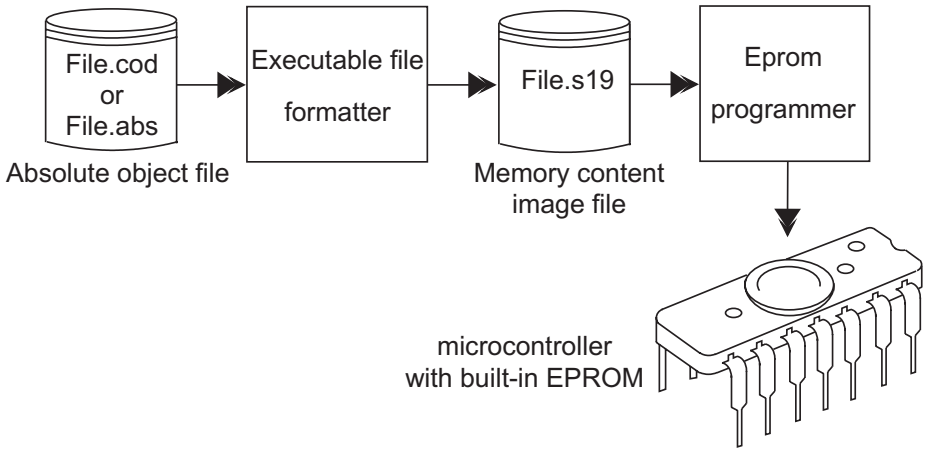
Each time a file has a successor with a date older than its predecessor, the corresponding tool is run to update the successor. This is done in an orderly fashion, from the top of the hierarchy (the source files) to the bottom (the absolute object file). Then, at a given time, all those source files that have been modified since the last run of the maker are reprocessed, and only those files. This both guarantees that no file will be forgotten in the updating process (provided that the control file correctly describes all the dependencies), and that only the necessary processing will be done, to save time.

The maker is an essential component of any set of programming tools. But unlike what was said about matching the same brand of assembler and linker, any maker may be used in a project, since all work the same way. The control file, in fact, uses a different syntax; but this does not affect the overall result. So, if you buy a new set of software tools for a microcontroller, you can still keep the maker you are accustomed to.

3.1.2.5 EPROM burners

EPROM burners, or programmers, are tools that have a hardware interface with a zero-insertion force socket that matches the characteristics of the specific EPROM chip or microcontroller with built-in EPROM. They transfer the contents of an absolute object file into the physical programmable memory. Burner software usually accepts several input formats. This gives some flexibility to the user and can allow using a burner from one manufacturer with software tools from another. Apart from accepting absolute object files generated by the linker in the same tool set, burners also frequently accept one or other of the more widely-used printable hexadecimal formats.

To use a hexadecimal format, either the linker must be able to output an absolute file in that format, or a code converter must be used that takes an absolute object file as an input, and produces a hexadecimal file as an output. Such converters are often supplied with in the software tool set. The use of these converters may save you the price of a new burner if you already own one for the right type of programmable component.



Programming a microcontroller

03-prom

3.1.2.6 Simulators

A simulator is a tool that pretends to run an absolute executable program. Like the other tools, it runs on the programmer's own PC. It interprets the machine code that is specific to the chosen processor or microcontroller, and outputs the results on the PC screen, without the need for actually building the application hardware or even obtaining a sample of the target microcontroller.

The simulator allows the program to be run step-by-step (instruction-by-instruction), allowing the programmer to watch the values changing in any register or memory location that result from the execution of the instructions. It also makes it possible to put breakpoints in the program, which are traps that stop the execution it reaches the address of the instruction where the breakpoint was put. This can be used to run some parts at full speed and then stop on reaching parts that are not yet fully functional, to avoid walking step-by-step through a lot of instructions that have already proven to be correct.

The progress of the execution can be followed on the screen in a special window that shows the source file, and a cursor indicating the next line to be executed. At the same time, other windows may display items like a range of data memory, the registers, the values of selected data in a selected format (byte, word, character string...).

The user is even allowed to alter the values of the registers of the core, of the input-outputs, or in memory, for example to correct a programming mistake, and continue the execution with

the correct values, so as to avoid having to edit and make the program each time an error is found.

In fact, simulation is limited to pieces of code that do not access the input-outputs, since this would require a complex program at simulator level to also simulate the behaviour of the external world that the application is supposed to interact with. But still, most simulators can correctly emulate the operation of the built-in peripherals, in particular the programmable timers, and can trigger interrupts whose servicing can be monitored on screen.

Simulation, though limited in its possibilities, has the advantage that it allows to you start debugging the application program before the hardware becomes available,. It can speed up the debugging process by ensuring that some pieces of code are already functional when the time comes to test and debug the hardware.

3.1.2.7 In-circuit emulators

In-circuit emulators are complex and expensive instruments that, for the application hardware, behave exactly like the intended microcontroller would do. From the user's point of view, an emulator is an inspection device that shows the calculations in progress in the application hardware and the state of the input-outputs, as well as the contents of the memory at any time. It is the electronic equivalent of a spy in the real world: an intermediate entity that, while playing its role correctly in the real world (the application hardware), at the same time informs an external intelligence (the programmer) of everything that happens in that world.

An emulator offers exactly the same features as a simulator does, in terms of running an application either at full speed or step-by-step, setting breakpoints, examining registers, memory and input-outputs. The major difference is that while running the program, the microcontroller actually interacts with the external world, producing actions on the product to be tested, and acquiring real data that it can process in a realistic way.

The simulator and the emulator are so close together in functionality that the manufacturer endeavours to provide the same user interface, i.e. the same presentation on the screen and the same controls through the keyboard and the mouse for both tools, so that the user has very little additional learning to do when switching from the simulator to the emulator.

One last feature that is found on top-range emulators (the very expensive ones) is so-called real-time tracing. This is a hardware system that records all events on the buses (address, control and data) and attaches a time tag to them. The memory capacity is at least one thousand events, and can exceed ten thousand in the most expensive emulators. This allows you to run the program at full speed, record what is going on, then quietly analyze the record to see what happened. In real-time applications, this may be the only way to understand what happens and the problems that occur, as most of these applications do not permit step-by-step debugging, since the processor must follow the activities of the external world in real-time. In

this type of situation, its a difficult task for the programmer to complete the debugging of the application so that it can be validated.

3.2 C LANGUAGE

3.2.1 Why use C?

Many high-level language exist; some have been written specifically for a family of microcontrollers, while others are widely-used languages from the computer world that have been adapted to suit the needs of microcontrollers. Today, one language that prevails in microcontroller programming, that is C language.

The primary advantage of C is the ability to write very simple statements that actually involve hundreds of machine instructions. As an example, the small sample of assembly source code given in the paragraph about the assembly language has five lines. It could be replaced by the following line:

```
memset ( (void*)1, 0xFF, 100 ) ;
```

This example is not very impressive. However, a C-language statement like:

```
int UnitPrice, Quantity, Cost ;
float Discount ;


void main ()
   {
   Cost = (float)( UnitPrice * Quantity ) * ( 1 - Discount ) ;
   }
```

is easy to read and translates into about eighty instructions, without taking into account the subroutines that perform multiplication and subtraction. We can see at a glance the productivity gain we can expect from a high-level language.

C language was initially designed as the programming language that came along with the UNIX operating system. The power and relative simplicity of it made it spread to many different cores, until it became the standard programming language for microcontrollers.

The advantage of C is that it is a language that, though being very powerful, remains very close to the hardware. This is a key feature for microcontroller applications, especially where parts of the code are written in assembler. Being close to the hardware means the ability for the compiler to produce very optimized code by choosing the best instructions for a given job, when several instructions are available. In particular, setting a port pin high or low must not

take a long string of instructions, since the designer of the microcontroller has worked hard to provide efficient instructions to do the job.

C is a structured language, meaning that code can easily be divided into blocks that can readily become functions, or the body of looping or conditional statements, keeping the source code legible if the writer has taken some precautions in the layout of the source text and provided useful comments.

Saying that C is a powerful language means that a short statement, taking only a single line of source text, may perform complex operations, involving hundreds or thousands of basic instructions. Not only does this conciseness help keep the source text easy to read, but also it guarantees the correct execution of the code it hides, since this code has already been extensively tested by the manufacturer of the compiler.

The availability of C on many computers makes it possible for you write and test some parts of the program directly on your PC. Most of the time, if the language constraints have been adhered to, a piece of code that works on a PC will work straight off on the target microcontroller. This is a recommended method of development, since it allows you to start writing and testing the program before the hardware of the application has actually been built. The less debugging is done directly on the hardware, the quicker the development takes.

The portability of C is also a guarantee for the future. The microcomputer world evolves so fast, that it is not unusal to have to redesign an existing application for another microcontroller, or to reuse parts of the code of a previous application for a future one. In such an event, the C language allows you to reuse code with virtually no change from project to project. This feature alone, pays back the moderate investment the C compiler represents.

To remove any doubts you might have about the sincerity of these arguments and since nothing is perfect, the drawbacks of C are be listed here.

The first drawback is that C was originally designed for mainframe computers. In these machines, there is only one memory space, that contains both code and data. In the ST7, there is also a single memory space that contains code, data and input-outputs, but the code resides in ROM and data in RAM. Standard C has no provision for specifying the address ranges of these parts of memory.

The interrupt service routines may, and if possible, should be written in C, like any other part of the code. However, standard C does not know anything about interrupts.

Inputs and outputs are handled in a mainframe through the operating system. Thus standard C has no special provision for handling them.

These drawbacks would totally prevent you from using C for microcontroller appilactions if the various implementations of C available did not provide solutions for these cases. These solutions vary from implementation to implementation, and constitute the main cause of non-port-

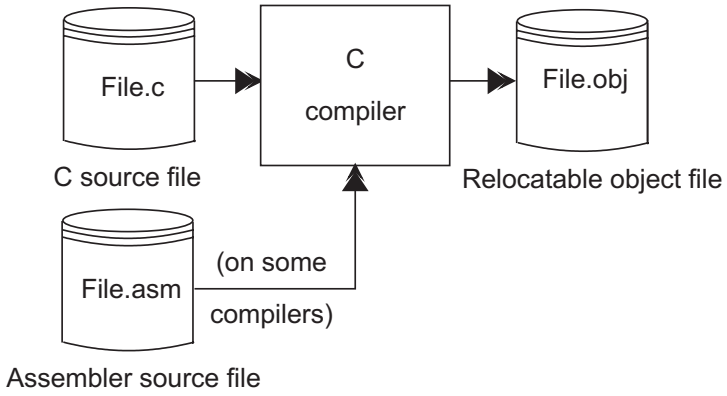
ability of programs to another machine. You will have to take account of this, especially when testing the code on your PC first as suggested above. This can in most cases be solved by a declaration file that is included in the project and that exists in two versions, or uses conditional compilation, to fill the gap between two different dialects of C. This point is actually the most tricky and that which requires most care from the programmer.

Having read these pros and cons, you have hopefully concluded that C is the language of choice for today if you want to save time and keep the fruit of your efforts over several years. We have kept the last argument for the conclusion, that will, undoubtedly, be decisive in the years to come.

The pressure for quality products is growing more every day. In some types of application, like medical apparatus and life-sustaining devices, equipment reliability has to be certified. More and more products, through ISO9000 standards or the constraints of product liability must be ready for quality assurance certification. Software quality assurance is a very difficult subject; it is not the purpose of this book to enter this field. But one thing is sure, only properly documented and structured software is likely to meet these quality assurance requirements.

3.2.2 Tools used with C language

Just like the assembler translates mnemonic language into machine code, the C compiler is a tool that translates a C-language source file into a relocatable object file. Here also, the application will be divided into files, also called modules, that are compiled separately. The whole group of object files is then linked to produce an absolute object file, the same as with assembler-generated object files. Actually, there is virtually no difference between an assembler-generated object file and one generated by a C compiler. This means that some modules may be written in C, and some in assembler; you can select the language according to the advantages and drawbacks of each language for each part of the application, as explained earlier.



C compiler invocation

03-comp

3.2.3 Debugging in C

Debugging in C is done the same way as explained for assembly language. However, the simulator and debugger screens designed for high-level languages can also show the progress of the execution directly in the C-language source text, while another window shows the corresponding assembly language statements. A cursor in each window keep the correspondence between both.

Another powerful feature offered by high-level language debuggers is the capability of displaying the data of selected variables according to their type, even for complex types like structures, arrays, strings, etc. This dramatically improves the productivity of the debugging phase, in the same proportion as choosing a high-level language does, in terms of shrinking the source code.

Thus the advantage of high-level languages is twofold: it reduces both the programming time and the debugging time. This is an invaluable benefit, as the resource that is probably the scarcest nowadays is time, even more than money. However, you are advised to check whether of the programming tools you selected fully support C language from one end to the other; just to mention a few essentials:

- A text-editor that provides features that are useful in C language, like auto-indenting, bracketed block selection, syntax highlighting, and so on;
- A compiler, an assembler and a linker that are really suited to working with the selected microcontroller, offering flexible addressable space allocation, efficient access to the ports,

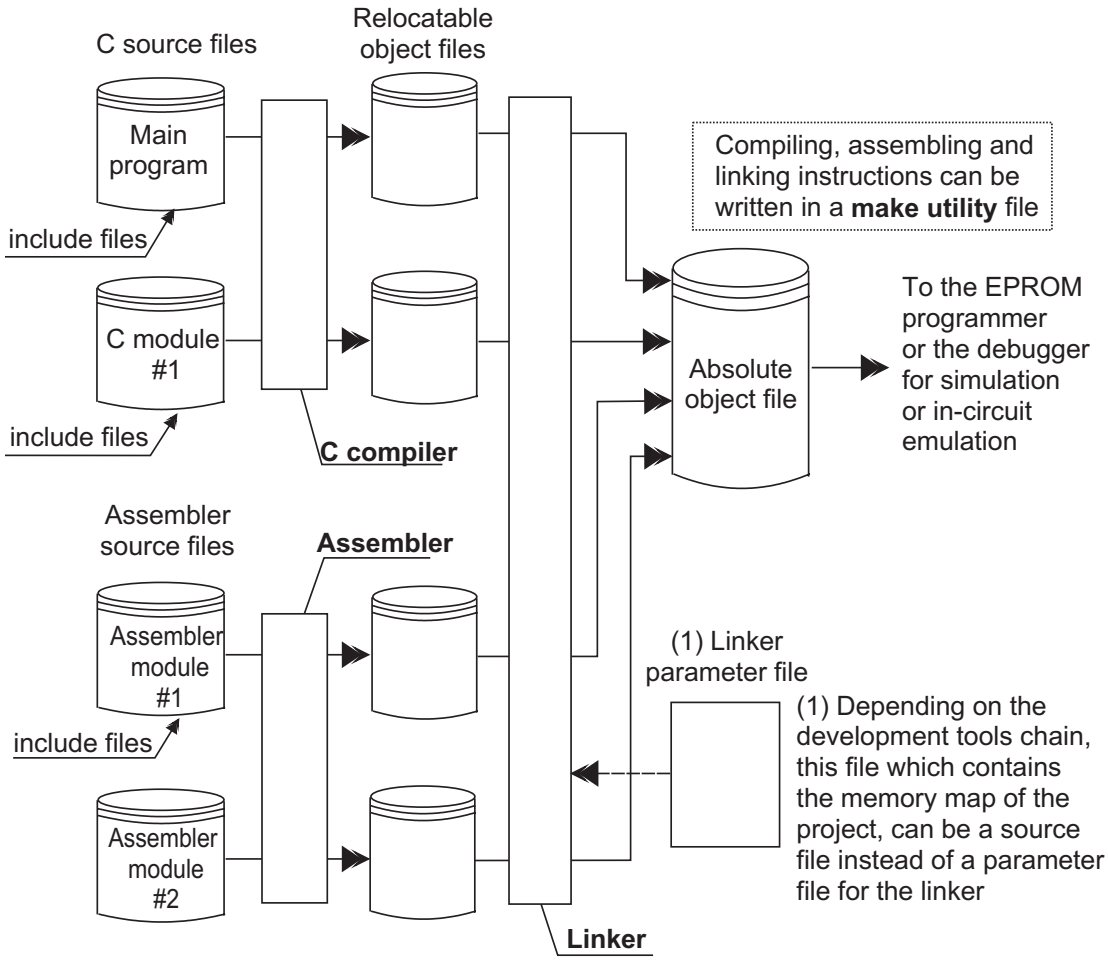
easy interrupt declaration and servicing in both assember and C language; efficient optimization that may be switched off at critical places;

- A simulator and a debugger that properly display the C source text, allowing you to easily change the values of variables or even code in memory to make patches, and that correctly synchronizes the timers during step-by-step execution.

These are just a few of the features that may make the difference between apparently equivalent competing products.

3.3 DEVELOPMENT CHAIN SUMMARY

The development chain discussed above can be understood more clearly using a diagram. The diagram below shows the various files involved in the development process as cylinders; the translators used to change one file type into another are shown as rectangles.


C source files
Relocatable object files
Main program
include files
C module #1
include files
C compiler
Compiling, assembling and linking instructions can be written in a **make utility** file
Absolute object file
To the EPROM programmer or the debugger for simulation or in-circuit emulation
Assembler source files
Assembler
Assembler module #1
include files
Assembler module #2
(1) Linker parameter file
(1) Depending on the development tools chain, this file which contains the memory map of the project, can be a source file instead of a parameter file for the linker
Linker

Typical software development tool chain

03-tchai

3.4 APPLICATION BUILDERS

An application builder is a graphical language that allows you to define the function of the program by manipulating icons on a computer screen. Each icon represents a standard function, like addition, integration, comparison, etc. Each icon has inputs and outputs, and by drawing lines on the screen between the output of one icon and the input of another, you connect these icons, that is, feed one program block with the output of the previous program block. This kind of programming tool is becoming popular in laboratory and plant automation since it allows you to build a complete application without leaving the block diagram conceptual level. This makes it a quick and easy-to-use programming tool for non-programmers.

An application builder for the ST6 and the ST7, the STRealizer, allows you to use graphical input techniques and build programs based on functional block diagrams. Version 2.2 of the STRealizer is available on the ST7 CD-ROM.

3.5 FUZZY-LOGIC COMPILERS

Fuzzy logic is a mathematical theory that has been applied to cope with problems that are at the border between logical and analog computation. It has lead to many articles and books to explain both its basic theory and various methods of implementing it using microprocessors. Although it is a very attractive theory, there is no evidence so far of a typical application domain, though quite a few claims have been made about robots, washing machines and vacuum cleaners, that have been said to be able to adapt their operation to the job they have to do. One example was a washing machine able to determine the proper amount of water needed for a certain weight of clothes--without using a scale to measure the weight.

Several fuzzy-logic compilers are available today, in particular one for the ST6 family, however there is none yet for the ST7.

4 ARCHITECTURE OF THE ST7 CORE

4.1 POSITION OF THE ST7 WITHIN THE ST MCU FAMILY

The STMicroelectronics range of microcontrollers is very wide, from proprietary architectures like ST6 to ST10 to second-source products like microprocessors for PCs and Digital Signal Processors.

The proprietary range can be summarized as follows:

Type	Word size	Main features	Typical applications
ST6	8 bits	Very low power consumption, 1.2 to 8 KB ROM, timer, ADC, watchdog timer and more depending on the subtype.	Appliances, home automation (suitable for direct power line supply)
ST7	8 bits	Industry-standard instruction set, 256 to 3K bytes RAM, 4 to 60 KB ROM, ADC, SPI, 16-bit timer, and more depending on the subtype.	TV remote control, car radio control, RDS decoder, etc.
ST9	8/16 bits	250 ns instructions (on a 16-bit word), many internal registers, powerful addressing modes, interrupt priority controller, DMA controller, plus a whole range of peripherals capable of complex processing. 16 to 128 KB ROM, more than 256 bytes RAM.	Automotive body applications and car radio.
ST10	16 bits	100 ns instructions (on a 16-bit word), 72 KB Flash EPROM, 10+ KB RAM, ADC, 16-bit timer, USART, and more.	Engine management systems, air bags, etc.

As the table shows, the ST7 is positioned towards the low-end. It provides an economical trade-off between speed and price and is suitable where moderate computational power is needed together with a low-consumption device, like in TV remote control transmitters.

Alhough it is a low-end product, because the ST7 combines the world's best-selling 8-bit instruction set with a host of added functions from a range of smart peripheral blocks, it is remarkably versatile for a microcontroller of its class. It has a wide choice of versions in order to minimize the component count of each specific application.

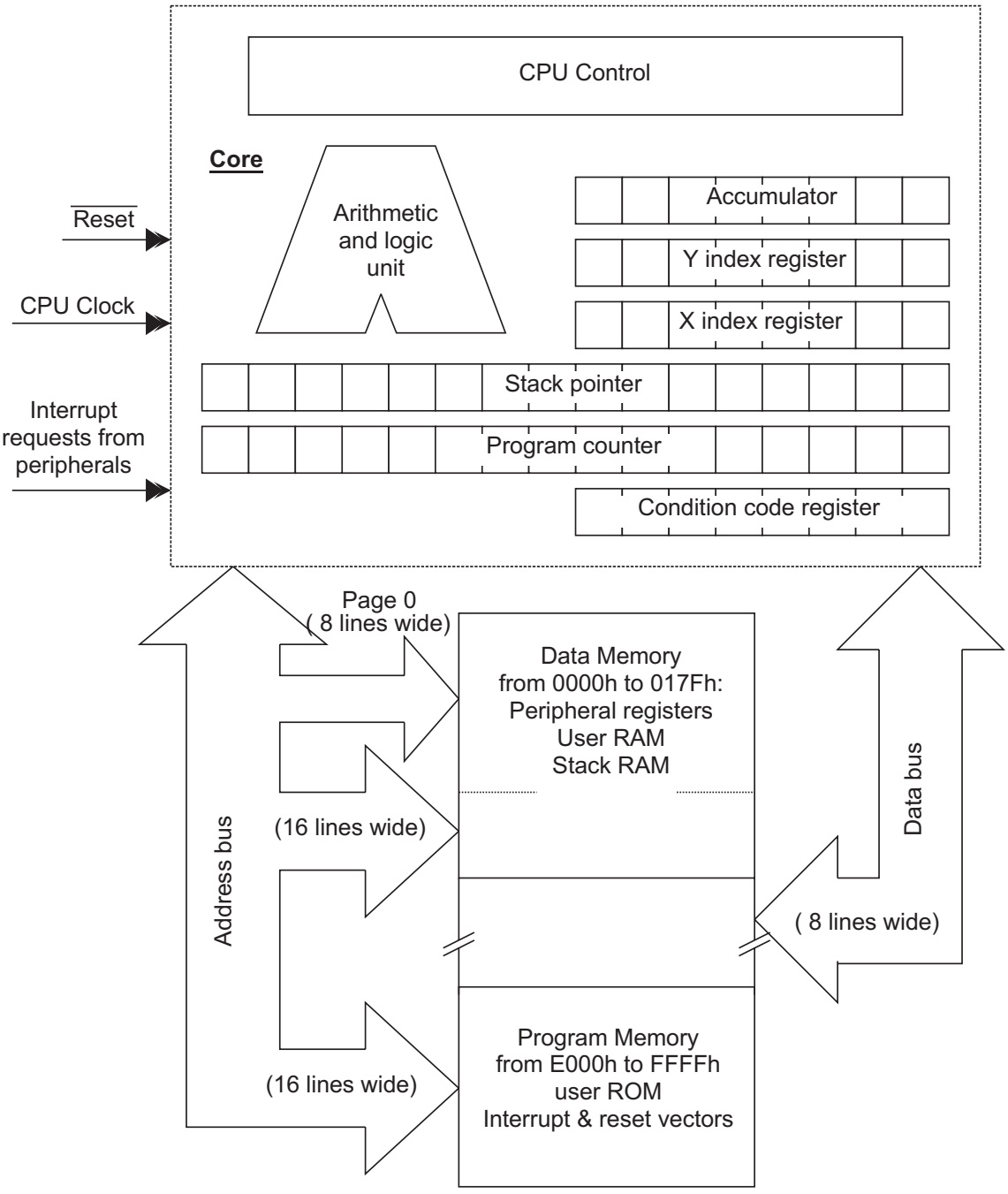
4.2 ST7 CORE

The ST7 core uses an extended version of an industry-standard instruction set. This set is both exhaustive (almost all instructions are available, with the exception of division) and fairly orthogonal (the instructions allow for most addressing modes). This makes the instruction set of the ST7 both clear and easy to use when programming in assembler.

For those of you already familiar with the standard instruction set, the extension resides mainly in the following two points:

- A second index register, Y, that can be used anywhere the X register is used (like indexed addressing), or like X, moved to any other register;
- The indirect addressing modes, that come in addition to the direct, long, and indexed modes.

The core as described here is limited to the processing unit, excluding the reset and interrupt circuitry. They will be described in the chapter dealing with the peripherals. The block diagram of the core and the addressing space is shown below:



The core and the addressing space of the ST72251

04-core

4.2.1 Addressing space

The ST7, as said earlier, is based on a Von Neumann architecture. This means that there is only one addressing space in which the program, the data and the input-output peripherals are mapped. The advantages are:

- Access to any byte (of program, of data or of input-output) using the same instructions
- No special instructions to access constant data in program memory or input-output
- Ease of programming, due to simplification resulting from the first two advantages

As a typical 8-bit processor, the address bus of the ST7 is 16 bits wide, and thus able to address 65,536 bytes. This is enough for most applications within the range of the ST7.

To improve the efficiency of the code, the addressable space is actually divided in two parts:

The addresses ranging from 0 to 255 (0FFh) are said to belong to page zero. An 8-bit address is enough to reach them.

The remainder, from 256 (80h) to 65,535 (0FFFFh), is accessed using 16-bit addresses.

The use of page zero can greatly improve the speed of the code if the most frequently accessed data are located in page zero. Also, in the ST7, all input-outputs are always located in page zero.

4.2.2 Internal registers

The core uses only six registers: A, X, Y, PC, SP and CC.

4.2.2.1 Accumulator (A)

The accumulator is the register where the arithmetic and logic operations are performed. To perform a two-operand operation between two values stored in memory, one of the values must first be moved to the accumulator, since the instruction code provides only one address. It must be moved back to memory when the operation is done.

The instructions that require only one operand, like INCrement, DECrement, COMPLement, COMPare, Test for Negative or Zero, Bit COMPare, and so on, can act on the accumulator, or directly on the data in memory, if so desired.

4.2.2.2 Condition Code register (CC)

This register holds several bits that are actually more or less independent from one another. These bits are set or reset (or left unchanged) after the execution of certain instructions. For example, if an addition produces a null result, the Z flag is set; otherwise, it is reset. If the result is negative, the N flag is set, otherwise it is reset and so on. The CC register remembers the conditions after each instruction, and these conditions are used by the conditional jump instructions. The CC is laid out as follows:

1	1	1	H	I	N	Z	C

Condition code register

04-ccreg

The leftmost three bits, indicated as ones, are not used. When read, they yield ones.

C bit

The C bit is the carry that is generated by an addition or subtraction. When adding 16-bit numbers, for example, we first add the two least-significant bytes together, then the most significant ones. If the result of the first addition yields a result greater than 255, it cannot fit in a byte. For example, adding 1200 and 6230 in decimal yields 7430. In hexadecimal notation, this is:

4B0 + 1856 = 1D06

Adding these numbers is performed in two steps. First, B0 is added to 56, yielding 06 with a carry of one. Then, 4 is added to 18, yielding 1C, and with the addition of the carry we get 1D.

The role of the C bit of the Condition Code register is to remember that carry between the two additions. The first addition is performed using the ADD instruction, and the second must use the ADC instruction (add with carry) that increments the result of the addition if the C bit is a one.

The instructions that affect the carry bit are mainly addition and subtraction, the shift and rotate instructions, and of course instructions that directly affect the CC register.

The instructions that use the C bit are ADC and SBC (subtract with carry), the rotate instructions, and some conditional jump instructions.

Z bit

This bit is set to one to reflect the fact that a value is zero, whenever the accumulator is loaded or after any arithmetic or logical operation. If the value is not zero, the Z bit is cleared. The instructions that affect or use the Z bit are the same as those for the C bit.

N bit

This bit is set to one to reflect the fact that a value is negative, in two's complement notation. A negative number has its most significant bit set to one; so the N bit reflects the most significant bit whenever the accumulator is loaded or after any arithmetic or logical operation. If the value is positive, the bit N is cleared. The value zero is considered positive. The instructions that affect or use the N bit are the same as those for the C bit.

I bit

This bit is the global interrupt mask. When this bit is set, all interrupts are ignored. However, if an interrupt request is present, clearing the I bit immediately triggers an interrupt.

H bit

This bit is a similar to the C bit, since it is set when a carry occurs. This time, it is the carry between the two nibbles of a byte. A byte is composed of two four-bit groups called nibbles. The H bit is set whenever an arithmetic instruction produces a carry between bit 3 and bit 4 of the accumulator.

This half-carry is used when performing decimal arithmetic. In this case, each nibble contains a BCD digit, so that the bit pattern for 23 in decimal is 00100011, and reads also 23 in hexadecimal. This kind of coding is called Packed BCD. To correctly add two packed BCD numbers, some correction is necessary, which is made possible by the H bit. Several cases must then be considered:

- First case: adding the numbers 23 and 52.
  Once coded in packed BCD, they read 23h and 52h. If we add these numbers, we expect to find 75. Actually, if we perform the ADD instructions on these numbers, we find 75h, as the rules for adding two binary numbers imply. We directly get the right answer in packed BCD. No half-carry has occurred.
- Second case: adding the numbers 23 and 59.
  Once coded in packed BCD, they read 23h and 59h. If we add these numbers, we expect to find 82. But if we perform the ADD instructions on these numbers, we find 7Ch, as the rules for adding two binary numbers imply. This is because 3 + 9 = C in hexadecimal. C is not an acceptable digit in BCD. However, it is easy to see that if we add 6 to the total (the difference between 15 and 9), we get 82h, which is also the right answer in packed BCD. No half-carry has occurred.
- Third case: adding 28 and 59.
  We expect to get 87. Once added as above, we get 81h. This time, a half-carry occurred. This indicates that we must add 6 to the result, giving 87h, which is the right answer.

To summarize, if the addition of two packed BCD numbers gives no half-carry and if the least-significant nibble is less than 0Ah, the result is correct as it is. Otherwise, adding 6 will correct the result.

The same thing applies for the most-significant nibble: if it has a value less than A, and there is no carry, the result is correct; otherwise, adding 60h will correct the result.

Complicated as it might seem, the handling of packed BCD avoids having to convert numbers back and forth between binary and decimal, although this may be easier in some cases.

4.2.2.3 Index registers (X and Y)

The index registers are meant to hold addresses, unlike the accumulator which is meant to hold data. The value stored in X or Y is involved in the effective address calculation in some addressing modes. The availability of two index registers allows for calculating and managing

two addresses as is needed in a memory-to-memory data move, with or without alteration in between. However, these registers may also be used to store temporary data.

4.2.2.4 Program Counter (PC)

The program counter is the register that controls the sequencing of the instructions. The program is written as a series of instructions, and these instructions are stored in consecutive cells of the program memory. The Program Counter contains the address of the next instruction to be executed. This instruction is read from memory, then executed, and the PC is incremented so that it then points to the next instruction in the sequence.

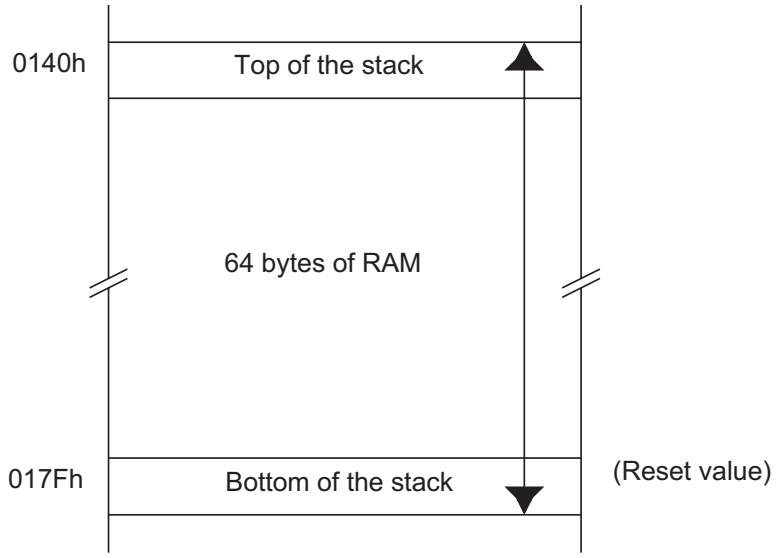
It is possible to alter the contents of the PC while the program is executing. In this case, the next instruction will not necessarily be the next in the sequence, but an instruction somewhere else in memory. Changing the course of the program is called jumping or branching.

Jump and Branch are the names of instructions that actually change the contents of the PC by setting it to a value specified with the instruction. Jumps may also be conditional, that is, the jump instruction effectively alters the contents of the PC if certain conditions are met. These conditions can be the values of one or more bits of the Condition Code Register. For example, the JREQ instruction changes the PC to the specified address if the Z bit of the CC register is set, otherwise the program continues in sequence.

Another kind of jump is the Subroutine Call, that first saves the address of the next instruction in sequence (the one that follows the jump instruction) before jumping. A special instruction, RETurn, retrieves this address and puts it into the PC. The next instruction executed is once again the one that follows the CALL instruction.

4.2.2.5 Stack Pointer (SP)

The stack is the part of the read-write memory where return addresses are stored by the CALL instructions and retrieved by the RET instruction.



Stack organization of the ST72251

04-stack

When a value is stored using the stack pointer, the stack is decremented, so that the next value stored will be placed at the address just below the previous one. This process of storing and decrementing the pointer is called Pushing, and can be done either by a PUSH instruction or by a CALL instruction that pushes the return address.

When the data is read back from the stack, the SP is incremented so that the next data retrieved will be the one situated at the address above the previous value retrieved. This is called popping the data, and can be done using the POP instruction or the RET instruction that pops one address off the stack and jumps to that address.

If several addresses are stored successively because several CALLs were executed successively, the first RET instruction will pop the last address pushed, the second RET will pop the one-but-last address, and so on. This feature provides for the nesting of subroutines, where the last called is the first exited.

Interrupts, being a kind of subroutine call, also use the stack to store the context of the interrupted process. Since interrupts occur at unexpected times, all the core registers must be saved on entering the interrupt service routine. This is performed automatically by the interrupt mechanism that pushes in order PC, X, A and CC. The return from interrupt must thus use a different instruction from the return from subroutine, where only the PC was saved. The instruction IRET is supplied for this purpose, and restores the initial values of these registers. It should be noted that the Y index register is not saved automatically. The industry-standard

core has no Y index register, so its instruction set does not take the Y index register into account. If needed, it must be pushed on entering the interrupt service routine using a PUSH Y instruction, and restored by putting a POP Y just before the IRET.

The stack pointer must be initialized at the start of the execution. The RSP instruction resets it to its bottom value, that differs from one variant of the ST7 to another depending on the number of registers provided at the beginning of page zero.

The value of the SP may be transferred to A, X or Y or set from these registers. This allows you to access the data in the stack or to save the stack pointer. This is useful for example for building multitasking kernels, as mentioned in Chapter 2 and illustrated in Chapter 7.

The PUSH and POP instructions mentioned earlier in this paragraph may be used to temporarily store a register that has to be reused later. This is very useful as the core has not many internal registers.

4.3 INSTRUCTION SET AND ADDRESSING MODES

4.3.1 A word about mnemonic language

In the text above, we have given examples that use the mnemonic language. For those of you that are not familiar with the mnemonic language of the ST7, here is a refresher. More details will be given in the paragraph that discusses addressing modes and in the chapter about the assembler.

The mnemonic language spans the gap between machines whose language is exclusively numeric, and humans who are more comfortable with letters and words. Unlike high-level languages that provide for complex concepts that must be translated into machine language using complex constructs, mnemonic language is easily translated into machine language since there is almost a word-for-word correspondence between numeric machine language and verbal mnemonic language.

Mnemonic language, also called assembly language, associates short names to the various objects the programmer uses. A translation program, called Assembler, translates these words into numbers. The words involved belong to the following classes:

- Labels
- Operation mnemonics
- Operand names
- Macro names
- Numbers
- Comments

From the categories above, numbers can be distinguished because they start with a decimal figure. For hexadecimal numbers that may start with a letter, there are two main conventions:

In the so-called Intel convention, a number will only be recognized as such if the first digit is a figure; for example, FAh is not considered a number; 0FAh is;

In the Motorola and and few other conventions, any hexadecimal number must start with a special character, like $ for Motorola; no ambiguity is possible then. The ST7 tools use this convention by default.

All other categories except comments are made of single words that begin with a letter. For example, HERE is a legal name for a label.

The vocabulary of the Operation Mnemonic category is defined by the manufacturer. It contains the operation codes specific to the microcontroller, commonly called opcodes. There are also other words that are not opcodes, but that have a similar function in the language. For that reason, they are named pseudo-ops.

Opcodes are named after the abbreviation of the function of the instruction. Some are obvious, like ADD; some abbreviated, like SUB (subtract); others are acronyms like TNZ (Test for Negative or Zero).

The pseudo-ops include commands to the assembly program, to direct it to make memory reservations, like DS (reserve Data Storage) or DCW (Define Constant Word), or other similar commands.

The labels are names that the programmer freely assigns to data in memory, constant values, or pieces of code. This improves the clarity of the source text. For example, if the programmer has reserved a byte in memory for the result of an analog to digital conversion, he uses the following statement:

```
Voltage:    DS 1                    ; Voltage read back from input
```

He can then use this name to load this value into the accumulator:

```
            ld A, Voltage           ; Get value into accumulator
```

Which is easier to read than the numeric sequence:

```
C6 01 24
```

That is the translation of the statement above, supposing the variable `Voltage` had been assigned to the memory address `124h`.

In the examples of source lines above, the text that follows the semicolon (;) is a comment. It is ignored by the assembler, and its sole purpose is to inform the human reader of the source text, when he later reads it.

The last category of words listed above is the macro name. A macro is a piece of text the programmer may define freely and which he can give a name. Inserting the macro name in the source text will replace that word by the whole predefined text, saving the programmer typing effort.

4.3.2 Addressing modes

The ST7 has many addressing modes. In addition to those inherited from the industry-standard architecture, there are those involving the second index register, `Y`, that duplicate the addressing modes available with the `X` index register, and all the indirect addressing modes.

Having a choice of addressing modes might seems surprising. Anyone might think that indicating the address of the source or destination of a data move should be a straightforward matter. Actually, there are several cases to handle:

- If the address of the byte to be read or written is known at the time the program is written, the direct addressing mode is used. This mode has two variants: short direct mode addresses page zero; long direct mode addresses the whole range.
- If the address of the byte is not known when the program is written, this means that the data has the form of a record, a string, an array or any other structure that holds complex data. Since the ST7 only processes bytes, it is necessary to process this kind of data byte-by-byte. Then, the address of the byte to be read is computed when the program executes, and it is this address that indicates which byte must be read. According to the structure of the data, one addressing mode or another might prove more convenient, or fast, or efficient. The ST7 has a choice of addressing modes that take the address from an index register (indexed addressing), or from the contents of a memory byte (indirect addressing), or a combination.
- Instructions that load constants into a register use immediate addressing, which means that the data is located just after the instruction code in program memory. This data is skipped to reach the next instruction.
- Jump instructions often branch to an address that is close to the address of the jump instruction. If the distance of the jump is within the range -128 to +127 from the instruction that follows the jump, relative addressing is efficient since the address of the destination of the jump is obtained by adding the single byte that follows the jump instruction to the current value of the Program Counter. This byte is called displacement. This instruction thus saves one byte of program memory. All conditional jumps use this addressing mode.

- The last mode, called inherent, means that the data involved in the instruction does not need to be designated by an address, such as the instruction that increments the accumulator.

The many indexed and indirect modes available are useful for translating programs written in C language, since these programs frequently use complex data structures.

4.3.3 Instruction set

The instruction set of the ST7 includes many instructions. They can be sorted in different ways. Here is a grouping by number of addressing modes available. This kind of sorting may seem arbitrary, but there are actually groups of instructions that have a common function and also the same set of addressing modes.

Table 2. Table of the number of addressing modes versus the instruction types.

Type of instruction							
No operand (system)	**Multipli-cation and branch (relative)**	**Push and pop**	**Load memory with index**	**Load and compare index**	**Single operand arithmetic and test**	**Load memory with accumulator and jump long**	**Two-operand arithmetic**
1 (inherent)	2	4	9	10	11	14	15
HALT IRET NOP RCF RET RIM RSP SCF SIM TRAP WFI	MUL BRES BSET BTJF BTJT CALLR JR*	POP PUSH	LD mem, X LD mem, Y	CP X, LD X, CP Y, LD Y,	CLR CPL DEC INC NEG RLC RRC SLA SLL SRA SRL SWAP TNZ	CALL JP LD mem, A	ADC ADD AND BCP CP A, LD A, OR SBC SUB XOR

4.3.4 Coding of the instructions and the address

The instructions are coded using bytes stored in program memory. One instruction takes one to four bytes according to its type and the addressing mode. These bytes are, in order:

The prefix byte (optional); the operation code (opcode), and one or two bytes of address (optional).

4.3.4.1 Prefix byte

The prefix byte is used to extend the range of instructions. The opcode being a single byte, no more than 256 combinations of an operation and an addressing mode may be coded. The original industry-standard instruction set only uses one byte for the opcode. The ST7 increased the choice of addressing modes so that it was no longer possible to code them all using a single byte. A prefix byte has been created. When this prefix is put before an opcode, it changes the addressing mode of the opcode. There are three prefixes:

- `PDY` (`90h`) means that the next instruction must use the `Y` index instead of the `X` index.
- `PIX` (`92h`) means that the next instruction must change its addressing mode (whichever it is) to the corresponding indirect addressing mode.
- `PIY` (`91h`) is a combination of the above: the addressing mode is indirect, and the index register used is `Y`.

4.3.4.2 Opcode byte

The opcode uses a bit-level coding to specify the type of operation to perform (add, subtract, jump, etc.) and the addressing mode used (direct, indexed, indirect, etc.) in a single byte.

The tables below summarize the available instruction codes. The lines are the value of the higher nibble of the opcode; the columns are the lower nibble. The grouping of the instructions is clearly visible.

		Low digit							
	High digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	0	BTJT m,0	BTJF m,0	BTJT m,1	BTJF m,1	BTJT m,2	BTJF m,2	BTJT m,3	BTJF m,3
	1	BSET m,0	BRES m,0	BSET m,1	BRES m,1	BSET m,2	BRES m,2	BSET m,3	BRES m,3
	2	JRA	JRF	JRUGT	JRULE	JRMC	JRC	JRNE	JREQ
short	**3**	NEG	-	-	CPL	SRL	-	RRC	SRA
A	**4**	NEG	-	MUL	CPL	SRL	-	RRC	SRA
X	**5**	NEG	-	-	CPL	SRL	-	RRC	SRA
short, X	**6**	NEG	-	-	CPL	SRL	-	RRC	SRA
(X)	**7**	NEG	-	-	CPL	SRL	-	RRC	SRA
	8	IRET	RET	-	TRAP	POP A	POP X	POP CC	-
	9	-	-	-	LD X,Y	LD S,X	LD S,A	LD X,S	LD X,A
immediate	**A**	SUB	CP A, #n	SBC	CP X, #n	AND	BCP	LD A, #n	-
short	**B**	SUB	CP A, m	SBC	CP X, m	AND	BCP	LD A, m	LD m,A
long	**C**	SUB	CP A, m	SBC	CP X, m	AND	BCP	LD A, m	LD m,A
long, (X)	**D**	SUB	CP A, m	SBC	CP X, m	AND	BCP	LD A, m	LD m,A
short, (X)	**E**	SUB	CP A, m	SBC	CP X, m	AND	BCP	LD A, m	LD m,A
(X)	**F**	SUB	CP A, m	SBC	CP X, m	AND	BCP	LD A, m	LD m,A

		Low digit							
	High digit	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
	0	BTJT m,4	BTJF m,4	BTJT m,5	BTJF m,5	BTJT m,6	BTJF m,6	BTJT m,7	BTJF m,7
	1	BSET m,4	BRES m,4	BSET m,5	BRES m,5	BSET m,6	BRES m,6	BSET m,7	BRES m,7
	2	JRNH	JRH	JRPL	JRMI	JRNM	JRM	JRIL	JRIH
short	**3**	SLL	RLC	DEC	-	INC	TNZ	SWAP	CLR
A	**4**	SLL	RLC	DEC	-	INC	TNZ	SWAP	CLR
X	**5**	SLL	RLC	DEC	-	INC	TNZ	SWAP	CLR
short, X	**6**	SLL	RLC	DEC	-	INC	TNZ	SWAP	CLR
(X)	**7**	SLL	RLC	DEC	-	INC	TNZ	SWAP	CLR
	8	PUSH A	PUSH X	PUSH CC	-	-	-	HALT	WFI
	9	RCF	SCF	RIM	SIM	RSP	NOP	LD A,S	LD A,X
immediate	**A**	XOR	ADC	OR	ADD	-	CALLR	LD X, #n	-
short	**B**	XOR	ADC	OR	ADD	JP	CALL	LD X, m	LD m, X
long	**C**	XOR	ADC	OR	ADD	JP	CALL	LD X, m	LD m, X
long, (X)	**D**	XOR	ADC	OR	ADD	JP	CALL	LD X, m	LD m, X
short, (X)	**E**	XOR	ADC	OR	ADD	JP	CALL	LD X, m	LD m, X
(X)	**F**	XOR	ADC	OR	ADD	JP	CALL	LD X, m	LD m, X

As said about the prefix, this table changes to either Y index, or indirect, or both according to the prefix byte. The letter m indicates «memory» in instructions such as LD m, X.

4.3.4.3 The addressing modes in detail

Immediate mode

When a register has to be loaded with a constant value that has been fixed in the program source, that value is determined in the program source text and must be stored in the program memory to make it fixed and permanent. In such a case, the most effective way to retrieve the value is to use the immediate addressing mode. In this mode, it is not necessary to supply the address of the data. The processor expects the data to immediately follow the opcode in the program memory. This saves both time and memory size.

Example: the instruction

```
LD A, #10h

        A is loaded with a constant value between 00h and FFh.
```

loads the accumulator with the value 10, not the value stored at address 10. The coding of the instruction is (in hexadecimal):

```
A6 10
```

This instruction takes two bytes of memory.

Direct short mode

There are two direct addressing modes: short and long. They are identical, except for the number of bytes of the address of the operand. Direct addressing means that the opcode is followed by the address of the data in memory to be read or written. In the short version, the address is situated between the addresses 0 and 255, and only one byte is used for it.

Example: the instruction

```
LD A, 10h

      A is loaded with the value stored in an absolute memory address
        in zero page.
```

loads the accumulator with the value stored in memory at address `10h`. The coding of the instruction is (in hexadecimal):

```
B6 10
```

This instruction takes two bytes of memory.

Direct long mode

This mode works like the direct short mode, except that the full 16-bit address is supplied. The address then takes two bytes instead of one, lengthening the instruction by one byte. This is why direct short addressing mode should be used as much as possible to speed up execution and save memory space. This means that the most frequently accessed data must be placed

in memory below address 100h. The assembler automatically takes care of this, by selecting the appropriate addressing mode when possible.

Example: the instruction

```
LD A, 1234h

     A is loaded with the value stored in an absolute extended memory
       address.
```

loads the accumulator with the value stored in memory at address 1234h. The coding of the instruction is (in hexadecimal):

```
C6 12 34
```

This instruction takes three bytes of memory. It should be noted that the most significant byte of the address comes first.

Indexed mode

In this mode, the contents of register X (or Y if the prefix is used) is used as the address of the data to read or write. As the index register is only 8-bit, the address is lower than 100h.

Example: if the register X contains the value 26h, the instruction:

```
LD A, (X)

       A is loaded with the value stored in a memory address in page
        zero pointed to by the chosen index register.
```

loads the accumulator with the value stored in memory at address 26h. The coding of the instruction is (in hexadecimal):

```
F6
```

This instruction takes only one byte of memory. This mode is used if the address of the operand is not known at assembly time and must be calculated according to some rule.

Indexed with short offset mode

This mode works like indexed mode, but the instruction is followed by a byte, called displacement or offset, whose value is added to the value in the index to get the effective address.

Example: to access byte 4 of a character string starting at 23h in data memory, we first load the X index with the value 4. Then, the instruction:

```
LD A, (23h,X)

       A is loaded with the value of the RAM address memory pointed
        to by the sum of the specified 8-bit offset and the contents
        of the chosen index register.
```

loads the accumulator with the value stored in memory at address 27h (23 + 4). The coding of the instruction is (in hexadecimal):

```
E6 23
```

This instruction takes two bytes of memory. The farthest address that can be reached is 1FEh (0FFh + 0FFh).

Caution: there must be no space on either side of the comma within the parenthesis. Example: (23h, X) is incorrect.

Indexed with long offset mode

This mode is similar to the indexed with short offset mode, but the offset is a 16-bit number that takes two bytes. It allows any address to be reached within the whole addressing range.

Example: to access byte 64h of a character string starting at 4523h in data memory, we first load the X index with the value 64h. Then, the instruction

```
LD A, (4523h,X)
```

```
A is loaded with the value of the RAM address memory pointed
to by the 16-bit sum of the specified 16-bit offset and the
contents of the chosen index register. The whole memory can be
reached.
```

loads the accumulator with the value stored in memory at address 4587h (4523h + 64h). The coding of the instruction is (in hexadecimal):

```
D6 45 23
```

This instruction takes three bytes of memory. If the sum of X and the offset exceeds 0FFFFh, the effective address rolls over zero. For example, if X contains 83h and the offset is FFC2h, the effective address will be 45h.

Caution: there must be no space on either side of the comma within the parenthesis. Example: (4523h, X) is incorrect.

Relative direct mode

This addressing mode is only used in jump instructions. The opcode is followed by a byte that represents the offset or the displacement of the destination of the jump, relative to the current value of the Program Counter. This displacement is a 8-bit signed number, so that the range of such a jump is limited to -128 to +127 from the instruction following the jump. This mode is efficient in conditional jumps, since the pieces of code that correspond to opposite cases (test was true or false) are often located close to each other. When longer conditional jumps are required, the solution is to do a relative jump to a location within the reach of a relative jump, where an absolute jump is made to the final destination.

These short-range jumps are usually called branches to contrast with the jump instruction that uses long addressing mode and that can jump anywhere in the addressing space. In the ST7, they are called relative jumps.

Example:

At address 1200h, the following instruction:

```
JRA 11F1h

   The absolute address supplied in the source code will be translated
        into a relative displacement by the assembler.
```

is coded as:

```
20 EF
```

The two bytes of the instruction occupy addresses 1200h and 1201h. Thus, the next instruction will be found at address 1202h. The offset is calculated from this displacement. The distance between 1202h and 11F1h is -11h. In 8-bit, two's complement, this is noted EFh. Adding EFh to 1202h yields 11F1, which is the address of the destination of the jump. It should be noted that although the jump is relative, the operand of the jump instruction is the destination of the jump. The assembler automatically calculates the difference. If the destination is out of reach, that is farther than 127 bytes in either direction, the assembler generates an error message.

Relative indirect mode

This mode is also only used for Jump Relative instructions. The opcode is followed by a byte that is the address in memory that contains the displacement.

Example: if the displacement EEh is stored in the memory byte at address 58h, the following instruction is stored at address 1200h:

```
JRA [58h]

    The value of the relative displacement is stored at the specified
       memory address in page zero
```

is coded as:

```
92 20 58
```

The three bytes of the instruction occupy addresses 1200h to 1202h. Thus, the next instruction will be found at address 1203h. The byte at address 58h is read, giving EEh or -12h that is added to the Program Counter, giving 1203h - 12h = 11F1h

In this mode, the operand of the jump instruction is the address of the displacement. This displacement must be supplied separately, if necessary by an expression that is calculated at assembly time. Example:

```
Displ dc.b THERE - HERE

          It is up to the programmer to supply the right expression
       for the displacement, but the calculation with the actual
       values will be performed by the assembler.
```

where THERE and HERE are labels respectively designating the destination of the jump, and the address of the instruction that follows the jump instruction. Here again, if the difference yields a number that exceeds the range of a one-byte value, the assembler generates an error message.

Indirect short mode

In this mode, the address that follows the opcode is that of a memory byte called a pointer that contains the address of the data to read or write. This allows the effective address to be formed by the result of a previous calculation. For example, if the pointer at address 23h contains 79h, and the byte at address 79h contains 0, the instruction

```
LD A,[23]
```

```
A is loaded with the value stored at the short address pointed
  to by the specified memory address in page zero.
```

loads the accumulator with the contents of address 79h (and not 23h) that is zero. The coding of the instruction is (in hexadecimal)

```
92 B6 10
```

where the first byte is the PIX prefix. This instruction takes three bytes of memory.

The indirect short mode is similar to the indexed mode, except that it is not necessary to load the index register with the address of the operand; it may be used directly where it resides in memory.

Indirect long mode

This mode is similar to the indirect short mode, but the effective address contained in memory is a 16-bit word. This allows the whole address range to be accessed. The pointer must still be placed at an address below 100h.

For example, if the pointer at addresses 23h and 24h contains 7954h, and the byte at address 7954h contains 0, the instruction

```
LD A,[23.w]
```

```
A is loaded with the value stored at the extended address pointed
  to by the specified memory address anywhere in memory.
```

loads the accumulator with the contents of address 7954h (and not 23h) that is zero. The coding of the instruction is (in hexadecimal):

```
92 C6 10
```

where the first byte is the PIX prefix. This instruction takes three bytes of memory. Please note the «.w» that indicates that the long indirect mode must be used.

Indexed with indirect short offset mode

This mode is similar to the indirect short mode, in that the address that follows the instruction is the address of a 8-bit pointer in memory that contains an address. The difference is that this address is not used right away; it is first added to the contents of the index register, and the sum is the effective address that will be read or written.

Taking the same example as for the indirect short mode, if the pointer at address 23h contains 79h, the index register X contains 5, and the byte at address 7Eh contains 0, the instruction

```
LD A, ([23],X)
```

```
A is loaded with the value stored at the address pointed to
by the sum  of the 8-bit offset stored at the specified short
address and the contents of the chosen index register. The
range of this mode is 0 to 1FEh.
```

reads the contents of the address 23h, that is 79h, and adds it to the contents of X, that is 5, yielding the value 7Eh. This value is taken as the effective address whose contents are read, giving is zero. The coding of the instruction is (in hexadecimal):

```
92 E6 23
```

where the first byte is the PIX prefix. This instruction takes three bytes of memory.

Indexed with indirect long offset mode

This mode takes a 16-bit index in memory, and adds its contents with the 8-bit contents of the index register, yielding the effective 16-bit address of the operand.

For example, if the 16-bit pointer at address 23h contains 7945h, the index register X contains 5, and the byte at address 794Ah contains 0, the instruction

```
LD A, ([23.w],X)
```

```
A is loaded with the value stored at the address pointed to
by the sum  of the 16-bit offset stored at the specified
extended address and the contents of the chosen index
register. Thus, the whole RAM memory space can be addressed.
```

reads the contents of address 23h, that is 7945h, and adds it to the contents of X, that is 5, yielding the value 794Ah. This value is taken as the effective address whose contents are read, giving zero. The coding of the instruction is (in hexadecimal):

```
92 D6 23
```

where the first byte is the PIX prefix. This instruction takes three bytes of memory.

4.4 ADVANTAGES OF THE ST7 INSTRUCTION SET AND ADDRESSING MODES

In many programming applications, data may have complex forms. To ease data handling, high-level languages have been created that simplify coding by allowing expressions like:

```
C[K] = A[I] + B[J]
```

Where A, B and C are arrays of numbers, and I, J and K the indexes to these arrays. The high-level language compiler translates this so as to read the Ith element of array A using the available machine-language instruction. If these are arrays of bytes whose base address is somewhere in page zero, the following instruction sequence can be used:

```
LD X, I     ; Set Index register to value of index I of array A
LD A, ([A],X); Get value A[I]
LD X, J     ; Set Index register to value of index J of array B
ADD A, ([B],X); Add value of B[J]
LD X, K     ; Set Index register to value of index K of array C
LD ([C],X), A; Put result into C[K]
```

This is only one of the many examples where powerful addressing modes help translate high-level languages efficiently. In this case, the whole addition is performed in 22 cycles, or 5.5 µs at 8 MHz, and consumes 12 bytes of code.

5 PERIPHERALS

5.1 CLOCK GENERATOR

The core of the ST7 is supplied with an internal clock that comes from the division of the oscillator frequency. This division rate is programmable, allowing you to select the best compromise between speed and power consumption.

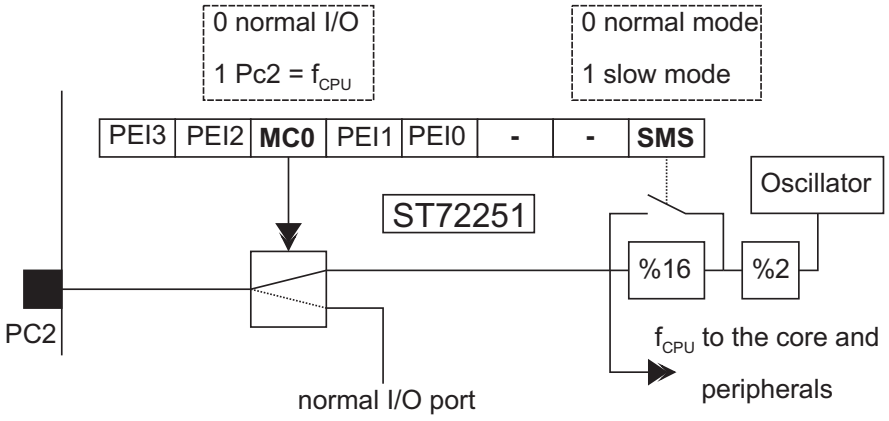
This choice is programmed using bits located in the Miscellaneous Register. This register also contains bits used for other purposes; their function will be detailed later in this chapter.

The Miscellaneous Register is laid out differently from one ST7 device type to another. We will describe only two variants here.

5.1.1 ST72251 Miscellaneous Register

The SMS (slow mode select) bit, when set, places the core and the peripherals in slow mode. This speed is 1/16 the normal speed, which is the oscillator frequency divided by 2.

The bit MCO (main clock out), when set, enables a clock signal with this frequency to be output on bit 2 of port C (PC2).



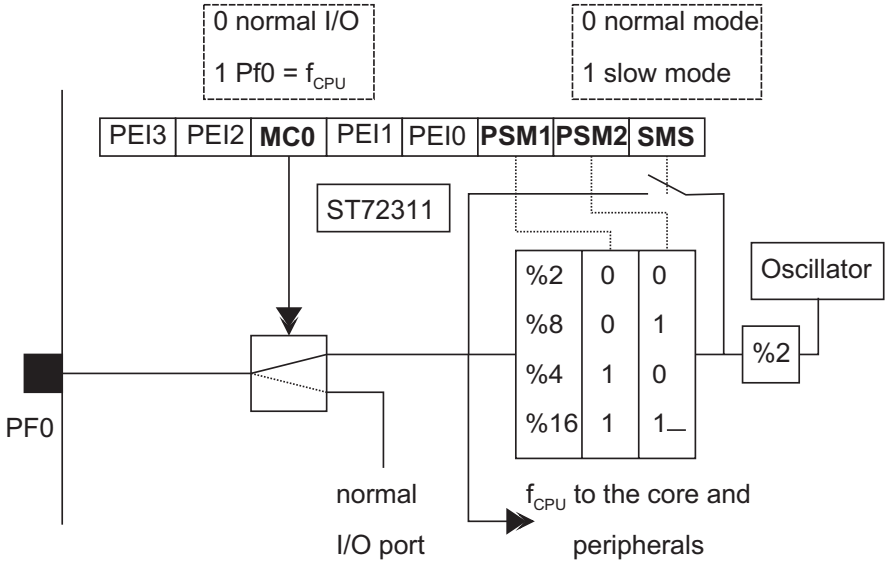
Main Clock Out and Slow Mode Select bits of the ST72251 Miscellaneous Register

05-misc1

5.1.2 ST72311 Miscellaneous Register

The SMS bit works the same way as in the ST72251 (see previous paragraph), but in addition, the two bits PSM1 and PSM0 (prescaler for slow mode) select the supplementary division rate between 2 and 16.

The bit MCO (main clock out), when set, enables a clock signal with this frequency to be output on bit 0 of port F (PF0).

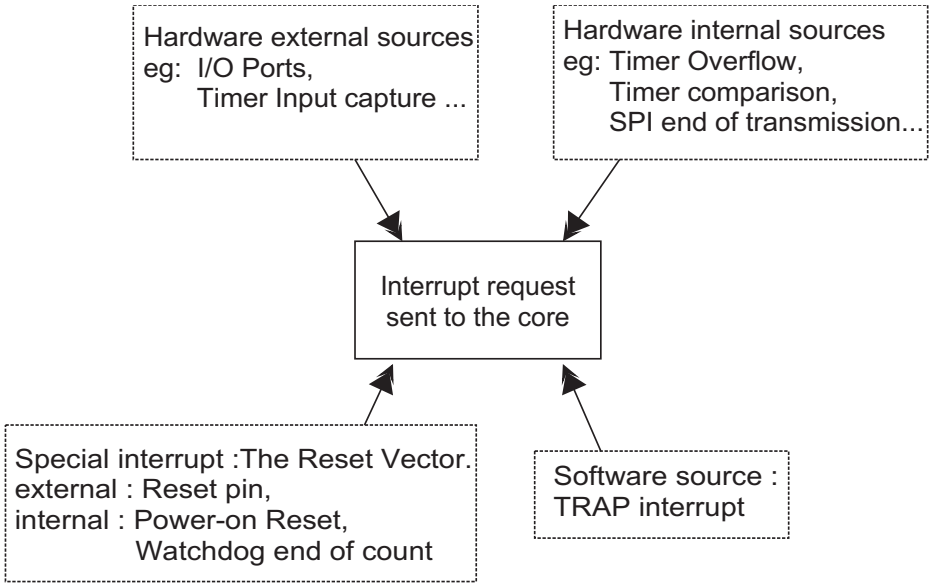


Main Clock Out, Slow Mode Select and Prescaler bits of the ST72311 Miscellaneous Register

05-misc2

5.2 INTERRUPT PROCESSING

Interrupt requests may be generated by several external or internal sources. Most peripherals of the ST7 can produce interrupt requests, such as the I/O ports, the timers or the SPI. The Interrupt request depends on the type of the peripheral: the I/O ports may have some bits configured to generate an interrupt, either on low-level, falling edge, rising edge, or both. The timer may request an interrupt on timer overflow, external capture or output comparison. The SPI may request an interrupt on end of transmission, etc.



The various sources of interrupts

05-sourc

5.2.1 Interrupt sources and interrupt vectors

The interrupt sources are summarised in the tables of the following paragraphs. There are compared the interrupt-related information for the ST72251 and the ST72311. It is interesting to see the differences.

5.2.1.1 Interrupts sources for the ST72251

Source Block	Description	Register	Vector address	Priority order
Reset	Reset	N/A	FFFEh-FFFFh	Highest
Trap	Software	N/A	FFFCh-FFFDh	
EI0	PA0-PA7	N/A	FFFAh-FFFBh	
EI1	PB0-PB7 PC0-PC5	N/A	FFF8h-FFF9h	
Not used			FFF6h-FFF7h	
SPI	Transfer complete	SPISR	FFF4h-FFF5h	
	Mode fault			
TIMER A	Input capture 1	TASR	FFF2h-FFF3h	
	Output compare 1			
	Input capture 2			
	Output compare 2			
	Timer overflow			
Not used			FFF0h-FFF1h	
TIMER B	Input capture 1	TBSR	FFEEh-FFEFh	
	Output compare 1			
	Input capture 2			
	Output compare 2			
	Timer overflow			
Not used			FFECh-FFEDh	
Not used			FFEAh-FFEBh	
Not used			FFE8h-FFE9h	
Not used			FFE6h-FFE7h	
I2C	I2C Peripheral interrupt	I2CSR1	FFE4h-FFE5h	
		I2CSR2		
Not used			FFE2h-FFE3h	
Not used			FFE0h-FFE1h	Lowest

5.2.1.2 Interrupt sources for the ST72311

Source Block	Description	Register	Vector address	Priority order
Reset	Reset	N/A	FFFEh-FFFFh	Highest
Trap	Software	N/A	FFFCh-FFFDh	
Not used			FFFAh-FFFBh	
Not used			FFF8h-FFF9h	
EI0	PA0-PA3	N/A	FFF6h-FFF7h	
EI1	PF0-PF2	N/A	FFF4h-FFF5h	
EI2	PB0-PB3	N/A	FFF2h-FFF3h	
EI3	PB4-PB7	N/A	FFF0h-FFF1h	
Not used			FFEEh-FFEFh	
SPI	Transfer complete	SPISR	FFECh-FFEDh	
TIMER A	Mode fault			
	Input capture 1	TASR	FFEAh-FFEBh	
	Output compare 1			
	Input capture 2			
	Output compare 2			
	Timer overflow			
TIMER B	Input capture 1	TBSR	FFE8h-FFE9h	
	Output compare 1			
	Input capture 2			
	Output compare 2			
	Timer overflow			
SCI	Transmit buffer empty	SCISR	FFE6h-FFE7h	
	Transmit complete			
	Receive buffer full			
	Idle line detect			
	Overrun			
Not used			FFE4h-FFE5h	
Not used			FFE2h-FFE3h	Lowest
Not used			FFE0h-FFE1h	

As shown in the rightmost column of these tables, these sources are prioritized. This means that if several interrupt requests are active at the same time, the interrupt controller will choose to service the highest priority request. When this interrupt is fully serviced, the next highest priority request will be serviced.

Please note that the tables are not identical: the number of vectors is different, and each vector groups a different set of interrupt causes, in particular the external interrupts (EI) are different. For other peripherals, such as the timer, SPI, etc. the grouping is identical, but the interrupt numbers and hence the priorities are different.

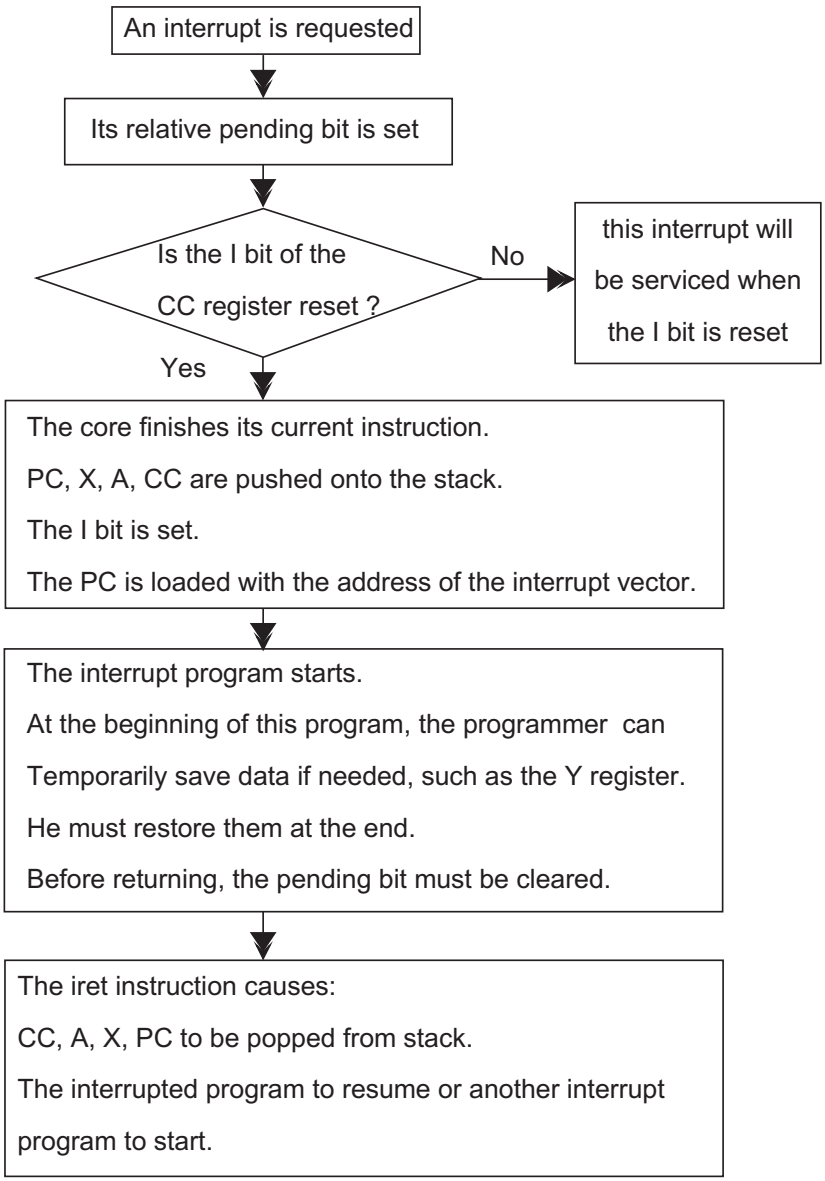
External interrupt inputs are further discussed in the paragraph about the parallel input-outputs.

5.2.2 Interrupt vectorization

When the core decides to grant an interrupt, it must know the address of the code to be executed when this event occurs. This is the purpose of the interrupt vectors.

The interrupt vectors are a table of 16-bit words in program memory that contain the address of the beginning of the various interrupt service routines.

According to the source of the interrupt (I/O, timer, etc.), the core fetches from a predefined location in memory, the address of the interrupt service routine especially written for processing that event. The vectors have a fixed location at the end of the addressing space. In addition to the interrupt vectors, the vector table also contains the reset vector that is fetched when the microcontroller is reset in order to get the address of the beginning of the main program.


An interrupt is requested
Its relative pending bit is set
Is the I bit of the
CC register reset ?
No
this interrupt will
be serviced when
the I bit is reset
Yes
The core finishes its current instruction.
PC, X, A, CC are pushed onto the stack.
The I bit is set.
The PC is loaded with the address of the interrupt vector.
The interrupt program starts.
At the beginning of this program, the programmer can
Temporarily save data if needed, such as the Y register.
He must restore them at the end.
Before returning, the pending bit must be cleared.
The iret instruction causes:
CC, A, X, PC to be popped from stack.
The interrupted program to resume or another interrupt
program to start.

The interrupt mechanism of the ST7 family

05-int

5.2.3 Global interrupt enable bit

RESET and TRAP (explained in the next paragraph) are non-miscible interrupt; the other sources of interrupt may be inhibited as a whole thanks to the I bit of the condition code reg-

ister. When this bit is set, no interrupts are generated. However, the interrupt requests are not forgotten; they will be processed as soon as the I bit is reset.

5.2.4 TRAP instruction

In addition to the hardware sources, a special instruction, TRAP, produces the same effect as an externally generated interrupt request, but under program control. Strange as this may seems (interrupts are provided to handle unexpected events, or at least, events for which the time of occurrence is not known), the TRAP instruction uses of the interrupt mechanism within the regular execution of the main program.

The trap instruction triggers interrupt processing regardless of the state of the I bit in the Condition Code register.

An example of the use of the TRAP instruction is the real-time debugger. When the user sets a breakpoint somewhere in the program, the debugger replaces the instruction at which the execution must stop with a TRAP instruction. The interrupt thus generated is processed by displaying on the screen the state of the microcontroller at that precise time. However, other uses of this instruction may be found as well.

5.2.5 Interrupt mechanism

5.2.5.1 Saving the interrupted program state

When the interrupt request triggers an interrupt, the first task of the core is to save its current state so as it will be able to restore it after the interrupt processing is finished. This is done by pushing all the core registers to the stack, namely the Program Counter, the X-register, the Accumulator and the Condition Code Register. It should be noted that the Y register is not saved, for compatibility with the standard instruction set which the ST7 adheres to. If needed, the Y register may be pushed explicitly to the stack at the beginning of the interrupt service routine.

At this point (and with the restriction above mentioned about the Y register), the interrupt service routine may execute freely. The status of the interrupted program is known and can be restored when needed.

To protect the interrupt service routine from other interrupt requests, the I bit of the Condition Code Register is then automatically set by hardware.

5.2.5.2 Interrupt service routine

When the processor has granted the interrupt request, and read the interrupt vector, it starts executing the interrupt service routine. This routine is merely a segment of program, written with exactly the same ease and constraints as the main program. It can be written using the same language and tools, or in any other language.

The interrupt service routine is supposed to take the appropriate actions that will vary according to the interrupt source. For example, if an Input bit has changed its state, the service routine may change the state of an output bit; if the interrupt was generated by the timer, this may produce the transmission of a byte by the SPI, etc. depending on the structure of the application as defined by the programmer.

Some interrupt vectors are common to several interrupt sources (e.g. the timers that have five sources). In this case, the first task of the service routine must be to check which source has requested the interrupt, before taking the appropriate action. This is done by testing the state of the various bits in the peripheral's Status Register.

Eventually, the service routine is finished. Then the core may return to the main program by ecuting the IRET instruction.

5.2.5.3 Restoring the interrupted program state: The IRET instruction

As described above, an interrupt service routine looks a little bit like a subroutine. Like in a subroutine, the return address is stored in the stack, and the execution of the RET instruction returns to the calling program.

Here, more things are to be done before returning to the interrupted program. All the core registers have been pushed on granting the interrupt request (except the Y register). They must now be restored, so that the execution of the service routine will not leave any trace in the core. This is the role of the IRET instruction.

The IRET instruction proceeds with popping off the stack all the data that had been pushed previously, namely the Condition Code (then the I bit is also restored), the Accumulator, the X register and the Program Counter.

From this time on, execution of the interrupted program resumes.

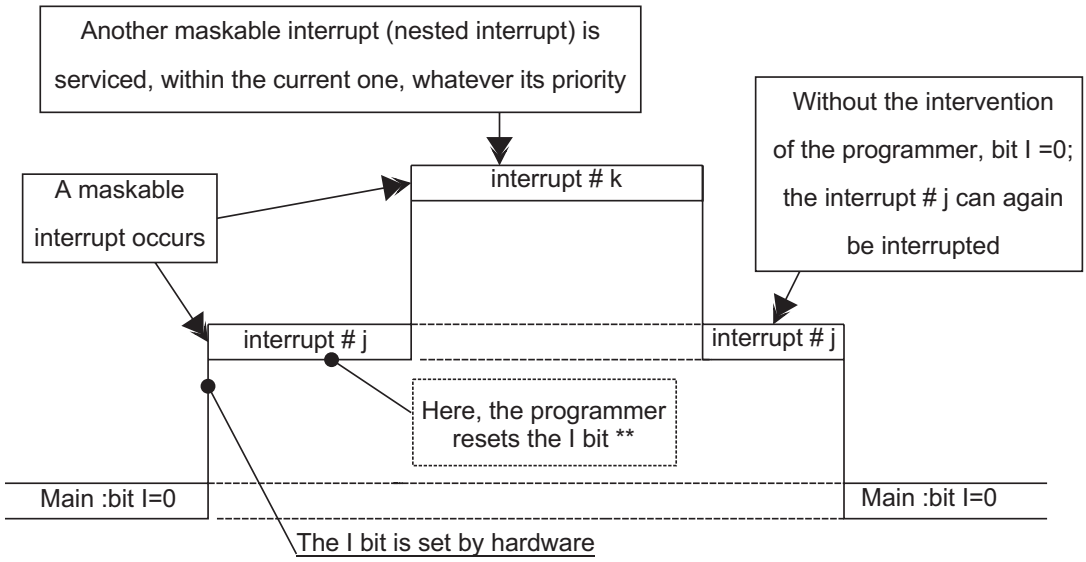
5.2.6 Nesting the interrupt services

Some interrupts may require the quickest service possible, for example if they signal the availability of some data that it volatile, that is, will not remain for a long time. In such a case, the fact that an interrupt service routine is not interruptible by default may be a problem, if the time to service this interrupt is longer than the allowable latency for the interrupt requiring a quick service. For so, it is possible to clear the global interrupt mask within an interrupt service routine.

At that time, one would probably like to restrain the interrupt request sources that may actually interrupt the service routine in progress. Some prioritizing, here also, is needed. But this is not the same priority mechanism as that mentioned above.

This priority mentioned about the interrupt sources is only taken into account when the core has to decide between two concurrent requests. It does not apply when interrupts are re-ena-

bled during the service of an interrupt. In this case, all interrupts are validated and the service routine may be itself interrupted by a request having a lower priority than its own. If the service routine must be interruptible only by certain requests and not others, the routine must first clear the interrupt enable flags of all peripherals that are not allowed at that time, then clear the global interrupt mask. It is wise to save the value of these flags before clearing them, if several interrupts may interrupt each other and each selects a different set of allowed interrupt requests.


Another maskable interrupt (nested interrupt) is serviced, within the current one, whatever its priority
Without the intervention of the programmer, bit I =0; the interrupt # j can again be interrupted
A maskable interrupt occurs
interrupt # k
interrupt # j
interrupt # j
Here, the programmer resets the I bit **
Main :bit I=0
Main :bit I=0
The I bit is set by hardware

** Note: before resetting the I bit, you should clear the pending bit that started the current interrupt, otherwise an endless recursive interrupt service call will be performed !

Nested interrupts

05-nest

This core is perfectly well able to handle nested interrupt servicing, with a limited nesting depth. However, care must be taken about two things:

The size of the stack. Each interrupt pushes five bytes on the stack, plus the interrupt service routine may push some bytes to protect any data in memory that may be used globally.

The problem of atomicity as explained in Chapter 2. If an interruptible service routine handles multi-byte data, it must take into account the possibility of being interrupted in the middle of a data read or write with the risk of data incoherence.

The applications described in Chapters 9 and 10 illustrate several uses of interrupts, in conjunction with various peripherals.

5.3 PARALLEL INPUT-OUTPUT PORTS

The purpose of the parallel input-outputs is basically to make binary signals either get into or out of the core. For the core and once initialized, they appear as a memory location that can be written or read. But in many cases, direct byte-wide input-output is not sufficient. Bit-oriented I/O is often required in systems driven by a microcontroller. This implies individually reading or writing any of the bits of each port. In addition, some of the external signals of the other peripherals (timers, UARTs, etc.) use an external connection without increasing the pin count by diverting some bits from the parallel I/O ports (alternate function).

The ST7 offers a very flexible feature on its parallel I/O that it shares with many other STMicroelectronics families: each bit can be independently configured as either an input with two options (with or without pull-up resistor), or an output with also two options (open-drain or push-pull).

Some pins are also available with high-current sink drivers, like Port A of the ST72251. These pin may sink up to 15 mA, and Port A has only one output configuration: open drain.

Selecting these options is done through registers that are associated with each port. They are memory-mapped, and are named Option Register (OR) and Data Direction Registers (DDR).

Some pins configured as input may also be connected to the external interrupt circuitry when the corresponding bit of the Option Register is set. This allows an interrupt request to be triggered when the state of the pin goes low, rises, or falls, as configured in the Miscellaneous Register already mentioned at the beginning of this chapter and detailed on the diagrams that follow. However some I/O pins can’t be interrupt inputs. See the tables below for the 72251 and the 72311 I/O configurations.

5.3.1 ST72251 I/O Ports

All bits of Port A are configurable as interrupt inputs when the corresponding bit of OR is set. The option for the edge and level of these inputs is set by the bits PEI0 and PEI1 (external interrupt polarity option) of the miscellaneous register. The external interrupt source is EI0 and its corresponding interrupt vector is in FFFAh-FFFBh.

All bits of Ports B and C are configurable as interrupt inputs when the corresponding bit of OR is set. The option for the edge and level of these inputs is set by the bits PEI2 and PEI3 in the same register. The external interrupt source is EI1 and its corresponding interrupt vector is in FFF8h-FFF9h.

Table 3. ST72251 I/O configuration

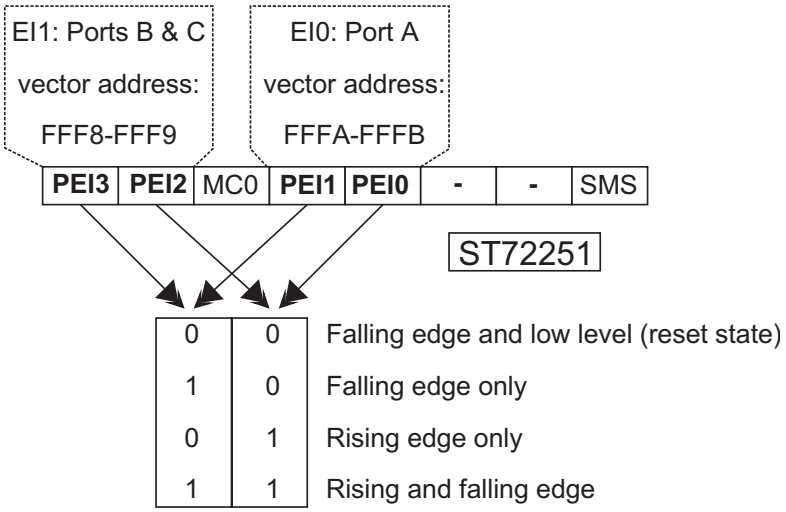
	Input configuration (DDR = 0)			Output configuration (DDR = 1)	
Port	OR = 0	OR = 1	External interrupt source, Polarity option bits	OR = 0	OR = 1
Port A: PA0-PA7	Floating	Floating with interrupt	EI0 PEI0-PEI1	True open drain	Reserved
Port B: PB0-PB7	Floating	Pull-up with interrupt	EI1 PEI2-PEI3	Open drain	Push-pull
Port C: PC0-PC5	Floating	Pull-up with interrupt	EI1 PEI2-PEI3	Open drain	Push-pull

In addition, some pins also serve as inputs for other peripherals, or may be held from their normal output function and be taken as the output pins of some peripherals if those peripherals are specially configured to do so by setting a bit in one of their control registers. As an example, pin PB0 is also used as the capture input of Timer A, and pin PC1 is the output compare 1 pin of Timer B if the OC1E bit of register TCR2 of this timer is set. The interaction with the pins and the required precautions are discussed, when needed, in the paragraph related to the corresponding peripheral.

Table 4. ST72251 I/O alternate functions

I/O pin	Alternate function 1		Alternate function 2	
PA4	SLC	(I^2C)		
PA6	SDA	(I^2C)		
PB0	ICAP1_A	(Timer A)		
PB1	OCMP1_A	(Timer A)		
PB2	ICAP2_A	(Timer A)		
PB3	OCMP2_A	(Timer A)		
PB4	MOSI	(SPI)		
PB5	MISO	(SPI)		
PB6	SCK	(SPI)		
PB7	SS	(SPI)		
PC0	ICAP1_B	(Timer B)	Ain0	(ADC)
PC1	OCMP1_B	(Timer B)	Ain1	(ADC)
PC2	CLKOUT	(Internal clock out)	Ain2	(ADC)
PC3	ICAP2_B	(Timer B)	Ain3	(ADC)
PC4	OCMP2_B	(Timer B)	Ain4	(ADC)
PC5	EXTCLK_A	(Timer A)	Ain5	(ADC)

Note: All pins of port C have two alternate functions.



External interrupt polarity Options EI0 and EI1 of the Miscellaneous register

05-EI251

5.3.2 ST72311 I/O Ports

Bits 0 to 3 of Port A and bits 0 to 2 of Port F of are configurable as interrupt inputs. The option for the edge and level of these inputs is set by bits PEI0 and PEI1 (external interrupt polarity option) in the miscellaneous register. The interrupt vector for Port A is FFF6-FFF7, and FFF4-FFF5 for Port F.

All bits of Port B are configured by the PEI2 and PEI3 bits in the same register. However, bits 0 to 3 are assigned to the vector at FFF2-FFF3, while bits 4 to 7 are assigned to the vector at FFF0-FFF1.

Note: PB5 to PB7 are not available on the smaller packages (J series).

Table 5. ST72311N I/O configuration

	Input configuration (DDR = 0)			Output configuration (DDR = 1)	
Port	OR = 0	OR = 1	External interrupt source, Polarity option bits	OR = 0	OR = 1
PA0-PA3	Floating	Pull-up with interrupt	EI0 PEI0-PEI1	Open drain	Push-pull
PA4-PA7	Floating		n/a	True open drain, high sink capability	
PB0-PB3	Floating	Pull-up with interrupt	EI2 PEI2-PEI3	Open drain	Push-pull
PB4-PB7	Floating	Pull-up with interrupt	EI3 PEI2-PEI3	Open drain	Push-pull
PC0-PC7	Floating	Pull-up	n/a	Open drain	Push-pull
PD0-PD7	Floating	Pull-up	n/a	Open drain	Push-pull
PE0-PE1	Floating	Pull-up	n/a	Open drain	Push-pull
PE4-PE7	Floating	Reserved	n/a	High sink capability	reserved
PF0-PF2	Floating	Pull-up with interrupt	EI1 PEI0-PEI1	Open drain	Push-pull
PF4, PF6, PF7	Floating	Pull-up	n/a	Open drain	Push-pull

Note: PA0-PA2, PB5-PB7, PD6-PD7 and PE4-PE7 are not available on the ST72311J version.

In addition, some pins also serve as inputs for other peripherals, or may be held from their normal output function and be taken as the output pins of some peripherals if those peripherals are specially configured to do so by setting a bit in one of their control registers. As an example, the PC2 pin is also used as the capture input of Timer A, and pin PC1 is the output compare 1 pin of Timer B if the OC1E bit in the TCR2 register of this timer is set. The interaction with the pins and the required precautions are discussed, when needed, in the paragraph related to the corresponding peripheral.

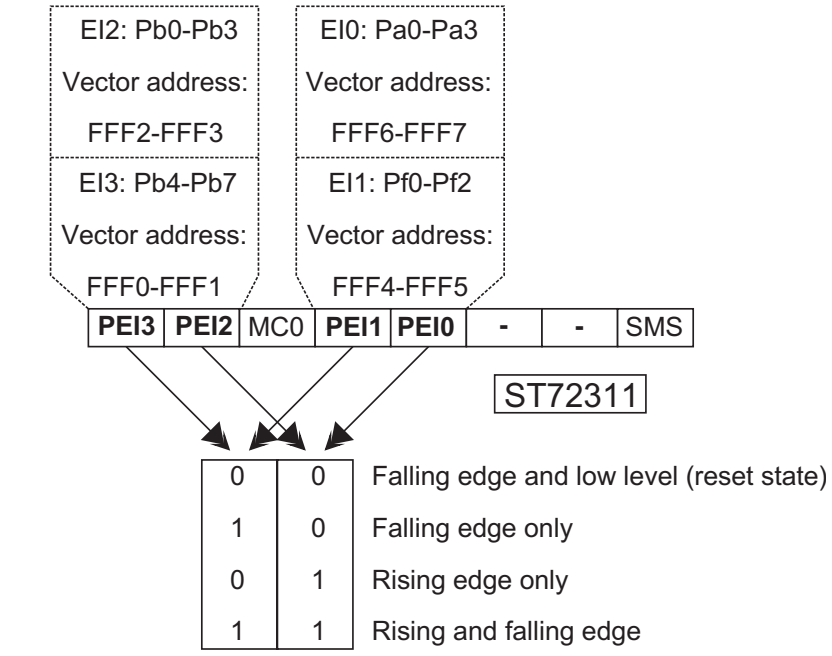
Table 6. ST72311N I/O alternate functions

I/O pin	Alternate function 1	
PC0	OCMP2_B	(Timer B)
PC1	OCMP1_B	(Timer B)
PC2	ICAP2_B	(Timer B)
PC3	ICAP1_B	(Timer B)
PC4	MISO	(SPI)
PC5	MOSI	(SPI)
PC6	SCK	(SPI)
PC7	SS	(SPI)
PD0	Ain0	(ADC)
PD1	Ain1	(ADC)
PD2	Ain2	(ADC)
PD3	Ain3	(ADC)
PD4	Ain4	(ADC)
PD5	Ain5	(ADC)
PD6	Ain6	(ADC)
PD7	Ain7	(ADC)
PE0	TDO	(SCI)
PE1	RDI	(SCI)
PF0	CLKOUT	(Internal clock out)
PF4	OCMP1_A	(Timer A)
PF6	ICAP1_A	(Timer A)
PF7	EXTCLK_A	(Timer A)

Note: None of the pins have two alternate functions, unlike the ST72251.

Since the pins can produce interrupt requests in some modes, you must take care, when programming the port configuration registers, to perform the transisitons in a particular order as specified in the transition map in the datasheet.

All the applications described in this book (Chapters 6, 7, 9 and 10) give examples of using the parallel ports in various combinations.



External interrupt polarity Options EI0, EI1, EI2 and EI3 of the Miscellaneous register

05-EI311

Note: Although the pins of the ports may be individually selected as interrupt inputs, the direction of the edge or the active level is selected by groups in the Miscellaneous Register. This implies that the electrical schematic takes this into account, and that signals that are rising-edge active be wired to a different group from those either falling-edge active or level-active

5.4 WATCHDOG TIMER

5.4.1 Aim of the watchdog

The watchdog timer is a safety device rather than a peripheral. Its purpose is not to handle external events, but to detect and correct internal malfunctions. However, it is implemented just like a peripheral, and this is why it has been included in this chapter with the peripherals.

A microcontroller, or any programmed machine, is not an electronic brain, in spite of how it was first introduced. Rather, it is an automaton that has a precise job to perform, taking into account events and conditions that are considered when the program is written. However, not all events can be taken into account; some occurrences are even neglected, since they are supposed to never happen. Rightly or wrongly, the code is thus made shorter. If, however, either because the programmer made a mistake or because a hardware failure produced an un-

foreseen event, the program may be fooled and the whole application may fail to work or even produce harmful actions.

To prevent this, two actions may be taken.

Write better code. Check what happens if a neglected condition arises. Lead the execution to a recovery routine in such an event. In short, take all precautions to prevent the program from crashing in any event. This is actually a requirement, not a choice. But still, things may happen that are totally out of the control of the author of the program. For example, an electromagnetic aggression or a power brownout to the product that is controlled by the microcontroller. Then, the proper working of the microcontroller may not be guaranteed and the system fails. This is when the watchdog can play its part.

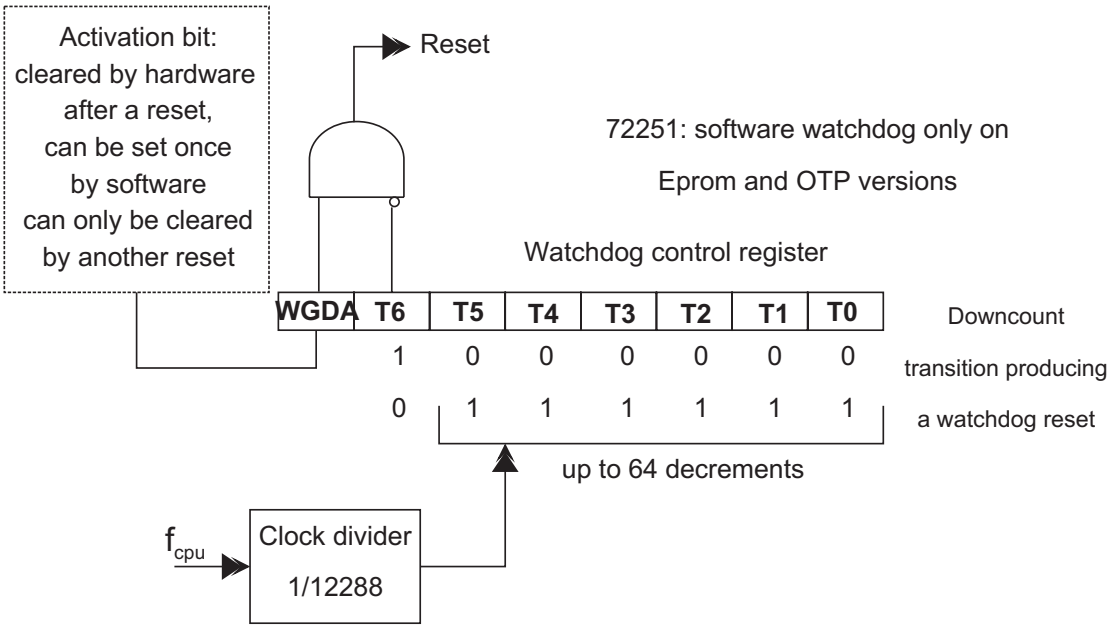
Methods of detecting processor failure by electronic means are virtually non-existant. A popular method relies on a timer that acts like an alarm-clock. The clock is wound up for a certain delay. If it has not been rewound before the expiration of this delay, the clock perform a hardware reset to the microcontroller.

It is up to the program to periodically rewind the clock (the watchdog timer) to indicate that it is still alive. Actually, it is not a full protection, since some parts of the program may crash while the part that has been elected to rewind the timer still functions. It is up to the wise programmer to find the program segment that is very unlikely to still work while some other part has crashed. Well implemented, this method gives rather good results. Of course, resetting the program is not a good way to recover from a fault, since the crash may have sent commands to the external world that are themselves faulty. The watchdog timer is actually a last ditch safety device, somewhat like a lifeboat in a shipwreck.

5.4.2 Watchdog Description

The ST7 watchdog timer is controlled by a register that includes two control bits (bits 7 and 6) and six time-setting bits.

The general control bit, bit 7, starts the watchdog activity if it is set to one. From that time on, it continues to work, even if one tries to reset it to zero. This is a safety measure that prevents the program from accidentally stopping it. The presence of this bit corresponds to what is commercially known as the “software activated watchdog”. This is the only option available for the 72251.

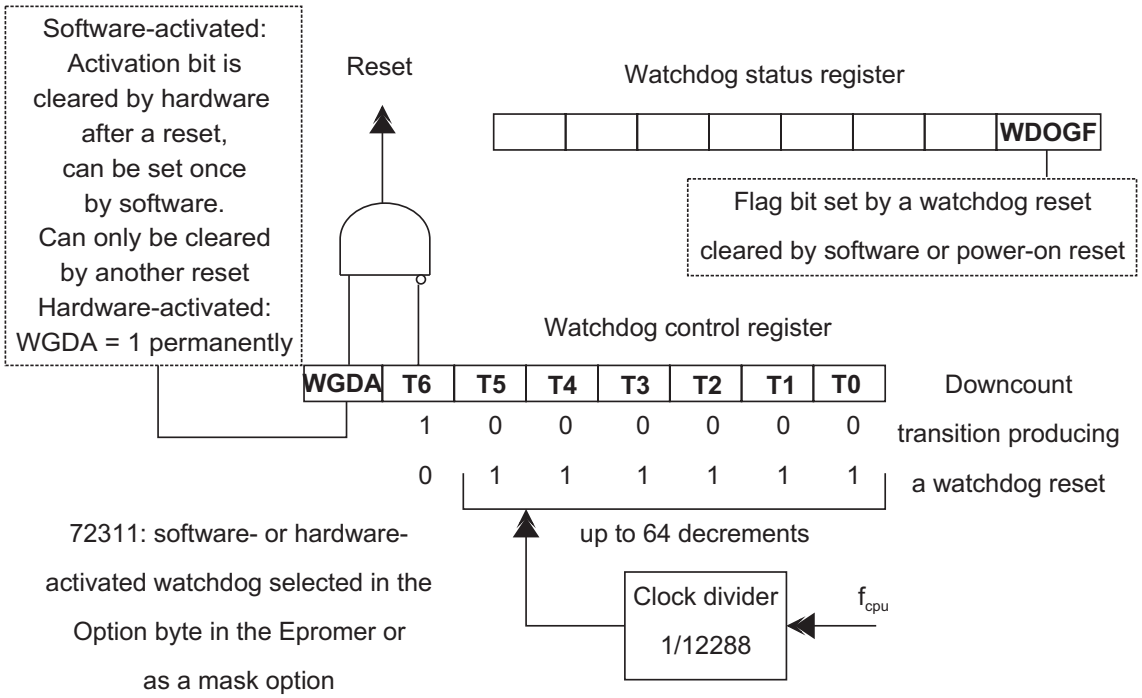


ST72251 Watchdog timer operating description

05-wdg1

In some ST7 family products, for example the ST72311, two options are available for the Watchdog: software-activated, as above, and hardware-activated, where the Watchdog is always active, regardless of the state of the MSB of the WDGCR register. This option is selected differently in the ROM version and the EPROM version.

- In the EPROM version, a so-called Option Register (OPTREG) has a bit that selects hardware- or software-activation. This register does not belong to the addressable space of the ST7, and is only accessible by the EPROM programmer.
- In the ROM version, the customer has the option to select software- or hardware-activation by specifying it on the order form.



ST72311 Watchdog timer operating description

05-wgd2

Bit 6 must be set high at all times. Setting it low immediately resets the microcontroller. This may be used to reset the whole application if needed under program control (e.g. a message received is coded as a reset request).

Bits 5 through 0 select the time-out value in increments of 12 288 machine cycles, which is roughly 1.5 ms at full speed. The time-out may be set in up to 64 increments, that is about 98 ms at full speed.

The watchdog control register is a read-write register, thus it is possible to read the time remaining in the watchdog counter. If the watchdog is not activated (by setting bit 7 to one for the software-activated version), it can be used as a real-time clock by simply reading its value.

To rewind the watchdog, it is sufficient to write a new value to it, keeping the two high-order bits high. It is possible to rewind the watchdog to different value, depending on the circumstances, to allow a shorter or longer time-out.

In the 72311, the watchdog has an extra register, the Watchdog Status Register (WDGSR) that only has one meaningful bit, WDOGF.

This bit is set whenever the last reset was triggered by the Watchdog. This allows the program to check whether the current start is a fresh one, or results from the recovering from an error condition.

5.4.3 Using the Watchdog to protect an application

The right value for the watchdog time-out is always difficult to find, except when the rewinding is performed in the service routine of a timer interrupt, that occurs at a fixed frequency. This is not the best way to use the watchdog, considering what has been said above, since it is likely that a timer interrupt will keep on being serviced even when the main program has crashed.

If the rewinding is done in the main program, it is often difficult to estimate the greatest possible interval between rewindings, since this time may vary depending on the various events that may occur. Here are two pieces of advice for anyone wanting use the watchdog timer:

- Do not activate the watchdog timer while debugging the program. Otherwise you may get some unexpected resets that may fool the in-circuit emulator.
- When the program is fully debugged, try several values by dichotomy. First set the value to half the maximum. If the reset occurs (which is detected by putting a breakpoint at the entry point of the main program), double this value and try again. If no reset occurs, take the value at midway and try again. Reduce the value this way as much as possible, each time using the program with all its features if possible. When you think you have found the smallest value that never produces a reset, multiply this value by a safety factor and keep it.

The safety factor depends on how much opportunity you had to actually test the program through all its paths, nooks and crannies. If this were actually possible, a factor of 1.5 may be sufficient. Otherwise, a factor of 2 or more is advisable. The terms of the trade-off are, on one hand, getting unwanted resets when nothing goes wrong, and in the other hand, reducing the efficiency of the safety device.

On models that have a WDOGF bit in the watchdog status register, the program may be written so that it behaves differently if the restart has been caused by a previous malfunction.

5.5 16-BIT TIMER

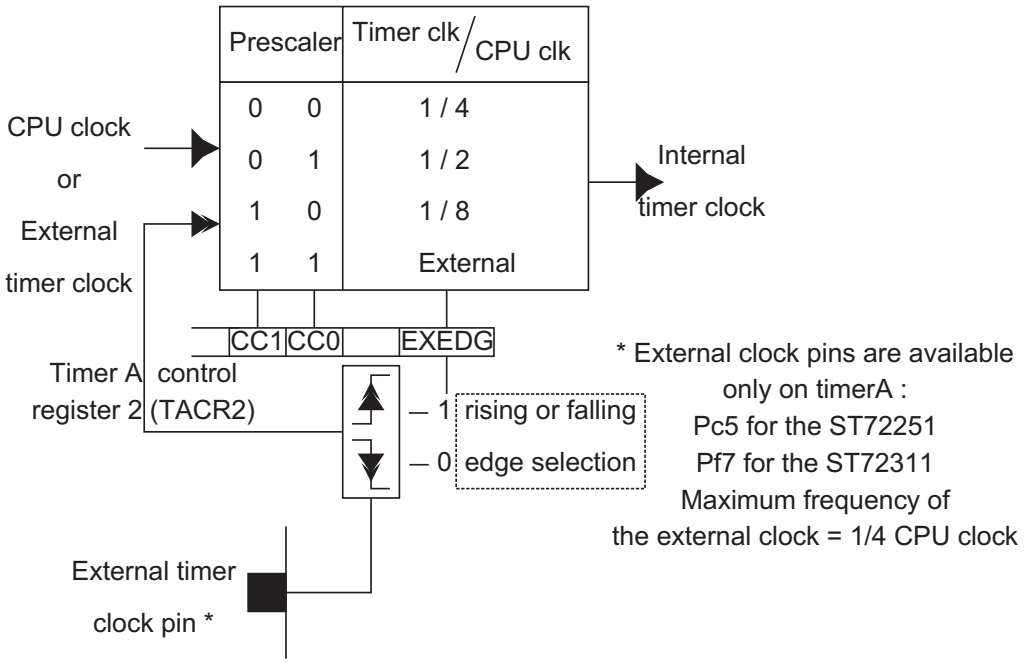
This peripheral is a powerful piece of hardware that illustrates the ideas put forth in the introduction of this chapter. Its purpose is to handle time-related events, such as pulse counting, frequency measurement, interval measurement, and pulse generation, either single or periodic. Good quality processing of such events usually implies a time accuracy in the microsecond range. This is out of reach for a low-end, 8-bit microcontroller. This is where the timer comes in to play: it does the time-related processing, and frees the core from stringent timing constraints.

5.5.1 Timer clock

Two clock sources are available for the timer: either the core clock, or an external clock supplied on a pin that takes up one port pin.

When the internal clock is used, the timer is actually fed with the core clock after passing through a frequency divider that can divide the core clock frequency by 2, 4 or 8. The core frequency is itself derived from the crystal oscillator by dividing it by 2 (normal mode) or a higher value that depends on the variant of ST7 considered (slow mode), as selected in the miscellaneous register (see the beginning of this chapter).

The external clock, when selected, changes one parallel input-output pin into a clock input. The frequency applied to the pin is directly fed to the timer with no predivision. The external clock is only available on Timer A on the ST72251 and ST72311.



Timer Clock selection

05-timck

To give an idea of the time resolution of the timer, the following table give the resolutions available with a 16 MHz crystal for all divider selections:

	Timer clock predivisor		
Core predivisor	1/2	1/4	1/8
1/2	0.25	0.5	1
1/32	4	8	16

5.5.2 Free running counter

The main component of the 16-bit timer is the Free-Running counter, called the CHR (Most Significant Byte) and CLR (Least Significant Byte) register. This is a binary counter that increments by one at each clock cycle, hence its name. It is possible to read the value of this timer, but it is only possible to reset it to its start value that is FFFCh, either under program control or automatically depending on the working mode selected.

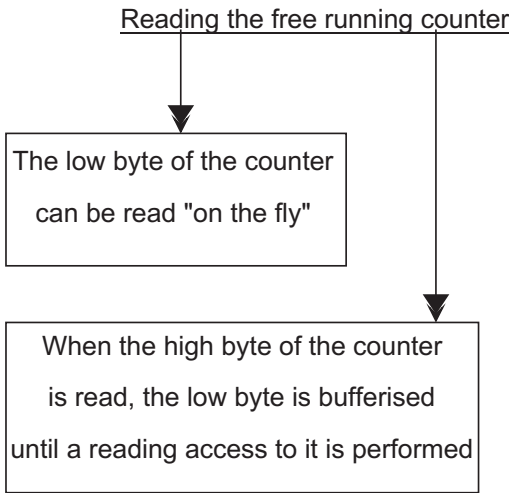
Each time the free-running counter overflows from FFFFh to 0000, the TOF bit (Timer Over-Flow) is set in the TSR status register. Resetting this bit involves reading the CLR register. In cases where it is necessary to read the free-running counter but it is not desirable to clear the TOF bit, the same free-running counter is accessible at another register address called ACLR where reading does not alter the TOF bit.

5.5.2.1 Reading the free running counter

The free running counter can be read at any time. However, since this is a 16-bit register, and the core is an 8-bit one, it is not possible to take a snapshot of the counter value at once. This can lead to data desynchronization problems, as mentioned earlier in this book.

To avoid this problem, the timer has a buffering feature that works as follows:

When the high byte of the counter is read, the value of the low byte is captured in a transparent latch. When the program reads at the address of the low byte of the counter, it actually reads the value previously frozen in the latch, and the latch is re-enabled. This means that when the low byte of the counter is read, for fast clock rates, the counter may not have the value read. The program must take this into account.


Reading the free running counter
The low byte of the counter
can be read "on the fly"
When the high byte of the counter
is read, the low byte is bufferised
until a reading access to it is performed

05-read

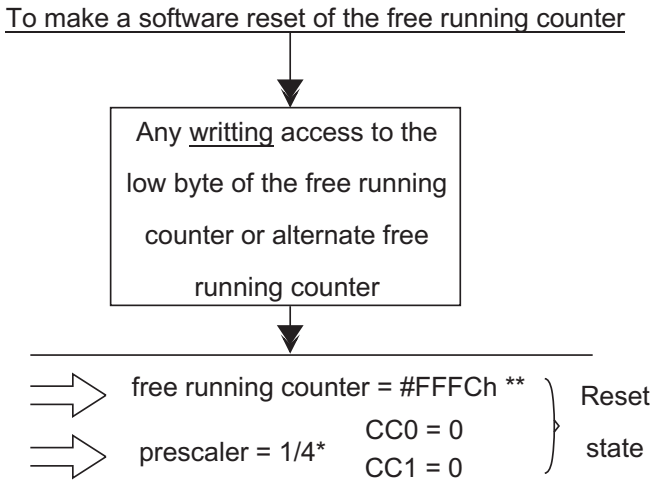
If the low byte is read again, the actual value of the low byte of the counter can be read. To benefit from the latching feature, it is necessary to read the high byte first. Conversely, reading the high byte one more time will always yield the same value as before: a read of the high byte must be followed by a read or the low byte.

This mechanism works identically for the CLR and the ACLR registers.

5.5.2.2 Resetting the free running counter

The free-running counter is actually a read-only register. However, it is possible to reset it by writing any value to the low byte of either CLR or ACLR (the value written is irrelevant). It is important to note that when reset, the counter is not set to zero, but FFFCh (or -4). This must be taken into account in the timing calculations.

The hardware reset also resets the timer to this value.


To make a software reset of the free running counter
Any writting access to the low byte of the free running counter or alternate free running counter
free running counter = #FFFCh **
prescaler = 1/4*
CC0 = 0
CC1 = 0
Reset state

* except CC0 & CC1, the other bits of the control register 2 are not affected

** the TOF flag goes high at the FFFFh - 0000h transition of the counter, that is to say 4 timer clock pulses after resetting the free running counter.

05-timrs

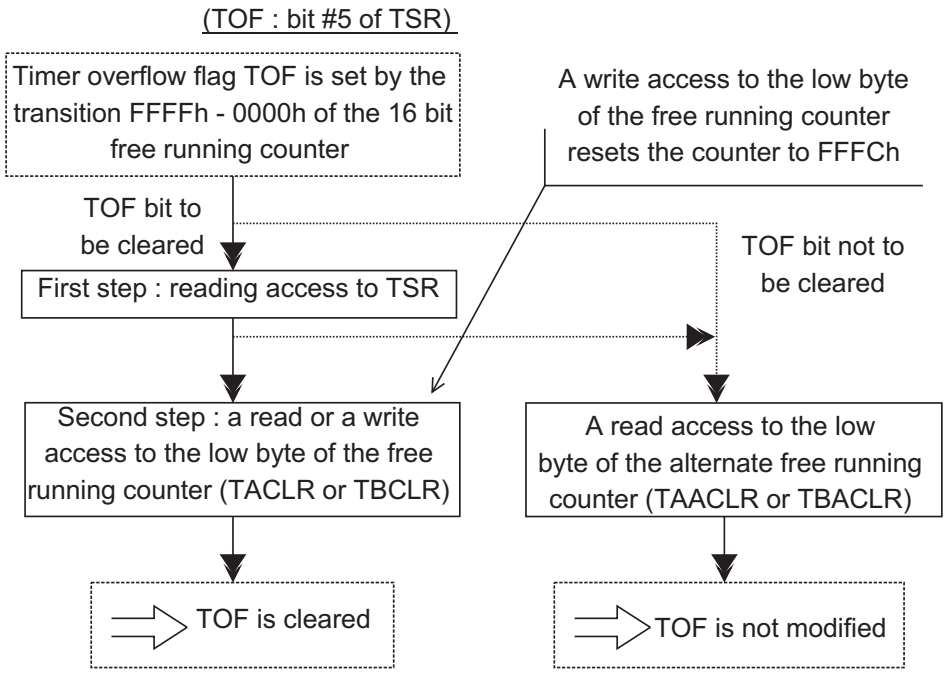
5.5.2.3 The TOF flag

The Timer OverFlow flag is a bit in the Timer Status Register. It is set each time the free-running counter overflows from FFFF to 0000.

It can be cleared under program control by the following sequence:

- A read to the TSR register, followed by
- A read or write to the CLR register.

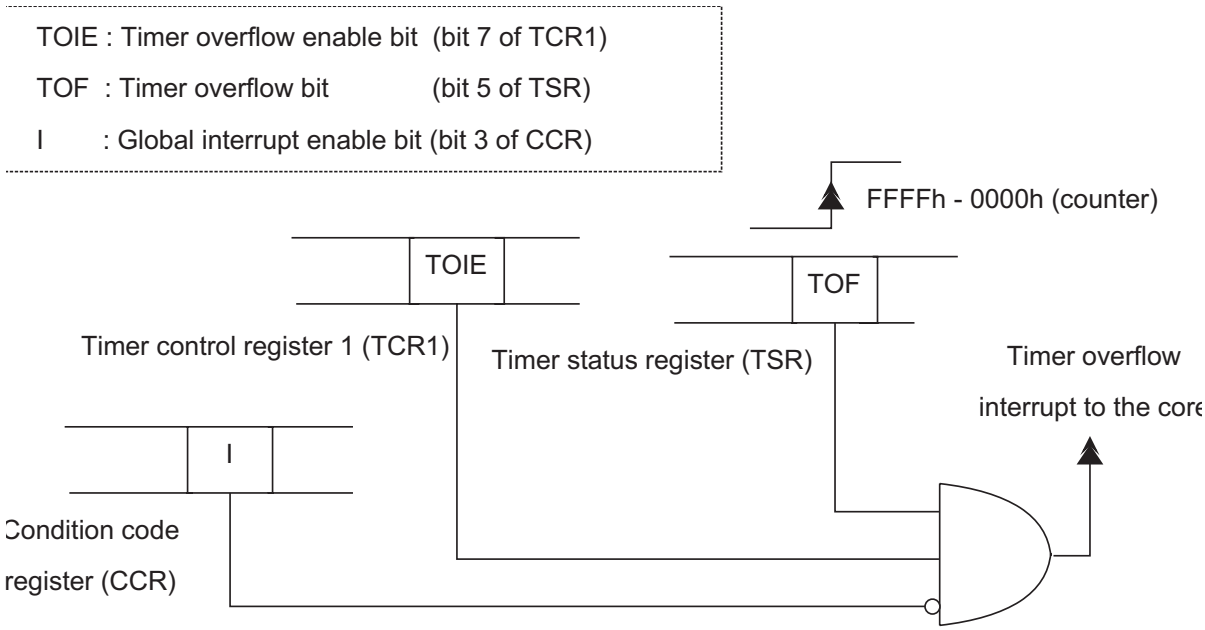
Any other sequence does not alter the TOF flag, especially using the ACLR instead of the CLR register. This is summarized in the diagram below:


(TOF : bit #5 of TSR)
Timer overflow flag TOF is set by the transition FFFFh - 0000h of the 16 bit free running counter
A write access to the low byte of the free running counter resets the counter to FFFCh
TOF bit to be cleared
TOF bit not to be cleared
First step : reading access to TSR
Second step : a read or a write access to the low byte of the free running counter (TACLR or TBCLR)
A read access to the low byte of the alternate free running counter (TAACLR or TBACLR)
TOF is cleared
TOF is not modified

Resetting the TOF bit in the Timer Status Register

05-tof

The timer overflow event is an interrupt source that can be either enabled or disabled by the TOIE bit of Timer Control Register 1. For this interrupt to be serviced, interrupt requests must be globally enabled by the I bit of the Condition Code Register:


TOIE : Timer overflow enable bit (bit 7 of TCR1)
TOF : Timer overflow bit (bit 5 of TSR)
I : Global interrupt enable bit (bit 3 of CCR)
FFFFh - 0000h (counter)
TOIE
TOF
Timer control register 1 (TCR1)
Timer status register (TSR)
Timer overflow
interrupt to the core
I
Condition code
register (CCR)

TOF interrupt mechanism

05-intof

The interrupt service routine must clear the TOF bit before returning, using the sequence described above. No automatic clearing is performed by the interrupt service mechanism.

5.5.3 Input capture operation

The counter, as discussed above, can be read on-the-fly by the program. This is not what the capture feature means here.

Note before continuing: each 16-bit timer has two capture channels, 1 and 2. In the following text, the letter i in the register names must be replaced by 1 or 2 accordingly.

The capture feature is a mechanism that takes a snapshot of the counter value at the time of the transition of an external signal applied to a pin. The capture mode is always enabled; the ICAPipin is shared with a parallel input-output port pin, and that port must be configured with the corresponding pin as an input to be able to use the capture input.

The IEDGi bit in the Timer Control Register 1 selects either the rising or the falling edge of the ICAPi pin as the active transition. When the transition occurs, the ICiHR-ICiLR pair contains the value of the counter at the time of the transition, plus one.

The capture event also sets the ICFi bit in the TSR register, and that bit can produce an interrupt request if the ICIE mask bit in the TCR1 register is set.

The clearing of ICFi is done by a sequence similar to that described for the TOF bit:

- Read the TSR register, and
- Either read or write the ICiLR register.

Only the ICIE flag is common to both channels. Thus, if both channels are used simultaneously, on interrupt, the TSR register must be read to determine whether the ICF1 or the ICF2 bit is set (or both). The ICFi capture flag bit(s) must be cleared by the interrupt service routine since no automatic clearing mechanism is provided.

The following table summarizes the registers and the bits involved.

- Timer A:

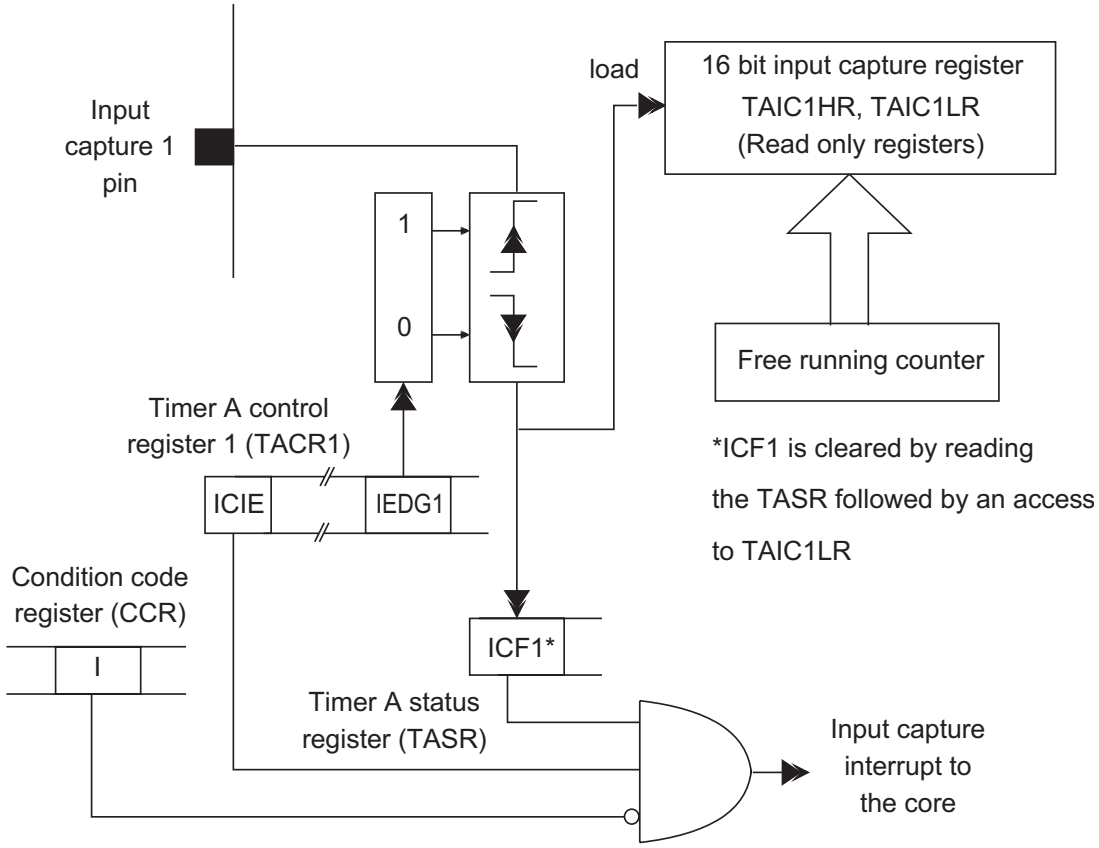
Channel 1	Channel 2	Function
TAIC1HR & TAIC1LR	TAIC2HR & TAIC2LR	Input capture registers (16-bit, read only access)
ICAP1_A PB0 on ST72251 PF6 on ST72311	ICAP2_A PB2 on ST72251 doesn't exist on ST72311	Input capture pin
ICF1	ICF2	Capture event flag in TASR
IEDG1 or TACR1	IEDG2 of TACR2	Capture Input Edge Selector.
ICIE		Common Interrupt Mask Flag in TACR1

- Timer B:

Channel 1	Channel 2	Function
TBIC1HR & TBIC1LR	TBIC2HR & TBIC2LR	Input capture registers (16-bit, read only access)
ICAP1_B PC0 on ST72251 PC3 on ST72311	ICAP2_B PC3 on ST72251 PC2 on ST72311	Input capture pin
ICF1	ICF2	Capture event flag in TBSR
IEDG1 or TBCR1	IEDG2 of TBCR2	Capture Input Edge Selector.
ICIE		Common Interrupt Mask Flag in TBCR1

The diagram below represents the timer A capture channel 1; the reader can transpose it for the other channel or the other timer by substituting the register, bit and pin names as in the table above.

ICF1 : Input capture flag 1 bit (bit 7 of TASR)
ICIE : Input capture interrupt enable bit (bit 7 of TACR1)
(common bit to both channels of the timer)
IEDG1 : Input edge 1 bit (bit 1 of TACR1)
I : Global interrupt enable bit (bit 3 of CCR)



Input capture and corresponding interrupt mechanism: diagram for channel 1, timer A

05-Capt

5.5.4 Output compare operation

The output compare is a feature that produces an event when the current value of the free-running counter matches the value of a register called Output Compare Register (OCnR, where n is 1 or 2 for the first of the second compare registers, respectively).

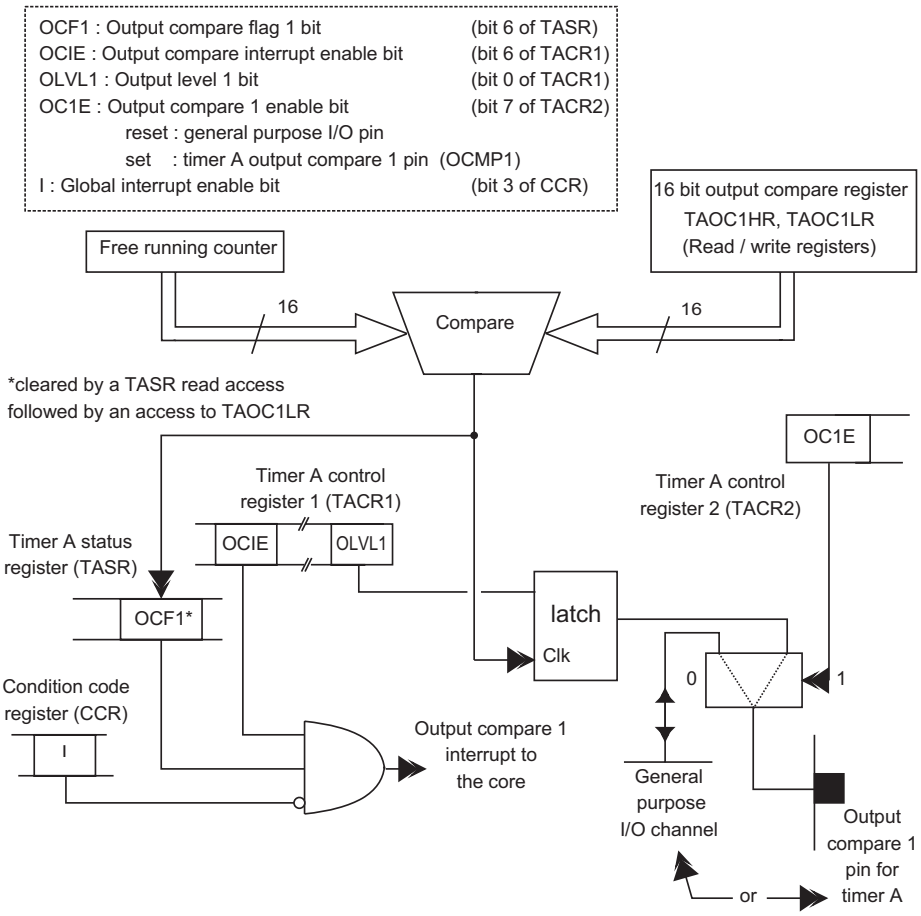
This event may produce various effects:

- An interrupt request;

- The change of the state of an output pin;
- The reset of the free-running counter (only in PWM mode).

The PWM mode will be studied later.

The block diagram of one of the compare circuits is the following:



Output compare and corresponding interrupt mechanism:
diagram for channel 1, timer A

05-comp

There are two such compare circuits for each free-running timer; the table below summarizes the register, bit, and pin names for both timers A and B and for compare 1 and compare 2 circuits of each.

- Timer A:

Channel 1	Channel 2	Function
TAOC1HR & TAOC1LR	TAOC2HR & TAOC2LR	Output compare registers (16-bit, read-write access)
OCMP1_A PB1 on ST72251 PF4 on ST72311	OCMP2_A PB3 on ST72251 doesn't exist on ST72311	Output compare pins (1 and 2)
OCF1	OCF2	Output compare event flag (bit in TASR).
OC1E	OC2E	Output pin enable bit in TACR2.
OLVL1	OLVL2	Level to be output on compare (bit in TACR1).
FOLV1	FOLV2	Force compare bit in TACR1
OCIE		Common Interrupt Mask Flag in TACR1

- Timer B:

Channel 1	Channel 2	Function
TBOC1HR & TBOC1LR	TBOC2HR & TBOC2LR	Output compare registers (16-bit, read-write access)
OCMP1_B PC1 on ST72251 PC1 on ST72311	OCMP2_B PC4 on ST72251 PC0 on ST72311	Output compare pins (1 and 2)
OCF1	OCF2	Output compare event flag (bit in TBSR).
OC1E	OC2E	Output pin enable bit in TBCR2.
OLVL1	OLVL2	Level to be output on compare (bit in TBCR1).
FOLV1	FOLV2	Force compare bit in TBCR1
OCIE		Common Interrupt Mask Flag in TBCR1

In the following text, we shall only consider the Output Compare 1 function of Timer A. The other combinations are exactly the same, if the letters A, B and the figures 1, 2 are replaced appropriately.

- The TAOC1HR-TAOC1LR pair is a 16 bit register whose value is continuously compared to the free-running counter. If the compare function is not used, it can be used for general purpose storage.
- When the match occurs, the OCF1 flag is set in the TASR register. This bit can only be reset by the following sequence: a read of TASR, then an access (read or write) to TAOC1LR.
- If the OCIE bit in the TACR1 register is set, this event triggers an interrupt request. The OCF1 bit is not cleared automatically, and must be cleared by the program as said above.
- If the OC1E bit is set in the TACR2 register, the OCMP1_A pin is driven by Timer A. When the OCF1 bit is set, this copies the level of the OLVL1 bit of the TACR1 register to the pin.
- The FOLV1 bit, when set, switches the timer into a mode where the output pin constantly reflects the state of the OLVL1 bit. The FOLV1 bit can only be set by software; to reset it, it is necessary to perform a hardware reset.

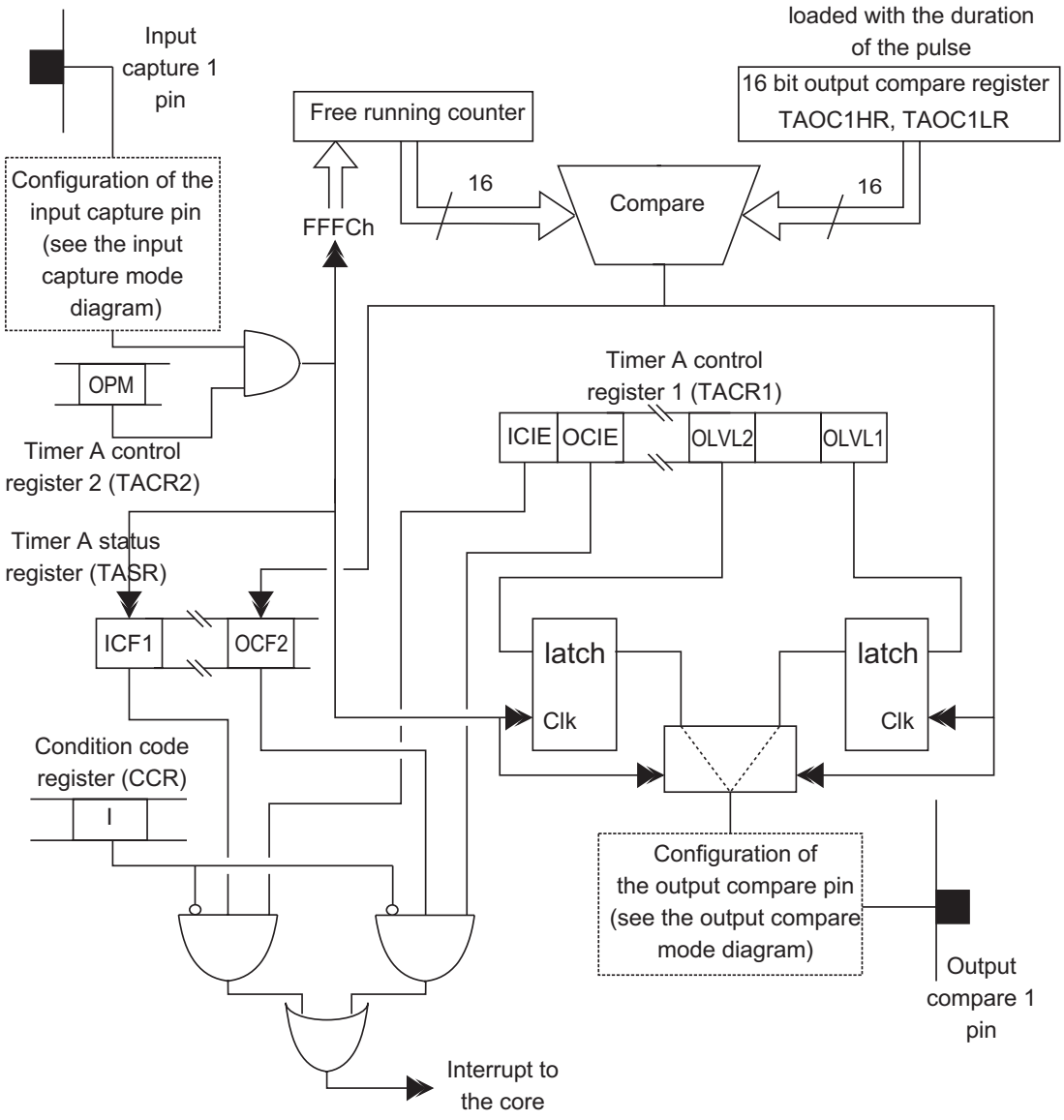
5.5.5 One-pulse mode

The input capture and output compare features are intermingled in One Pulse Mode. This mode is selected by setting the OPM bit in the TACR2 register (or TBCR2 for Timer B).

In this mode, an active edge on the ICAP1_A pin toggles the OCMP1_A output pin, then, after a predefined delay, this pin is toggled back to its initial level. This is the numeric equivalent of a one-shot multivibrator.

The settings for this mode are performed as follows:

- Set the TAOC1HR-TAOC1LR register pair to the number of ticks corresponding to the delay (this number depends on the clock frequency), minus 4.
- Set the OLVL2 bit of the TACR1 register to the state required for the output pin for the duration of the pulse, and OLVL1 of TACR1 to the complement of this state to terminate the pulse.
- Set the IEDG1 bit of TACR1 for the desired active edge on the input (0 for falling edge, 1 for rising edge).
- Set the OPM bit of TACR2 to enable the one-pulse mode.
- Set the OC1E bit of TACR2 to enable the output pin.



One pulse mode and corresponding interrupt mechanism: diagram for timer A

05-pulse

The input capture event both toggles the output and resets the free-running counter to FFFCh, and a successful match with the value in the register TAOC1R toggles the output back. This is why the compare register should be set to the calculated value minus 4.

The leading and trailing edges of the pulse can generate interrupts if desired:

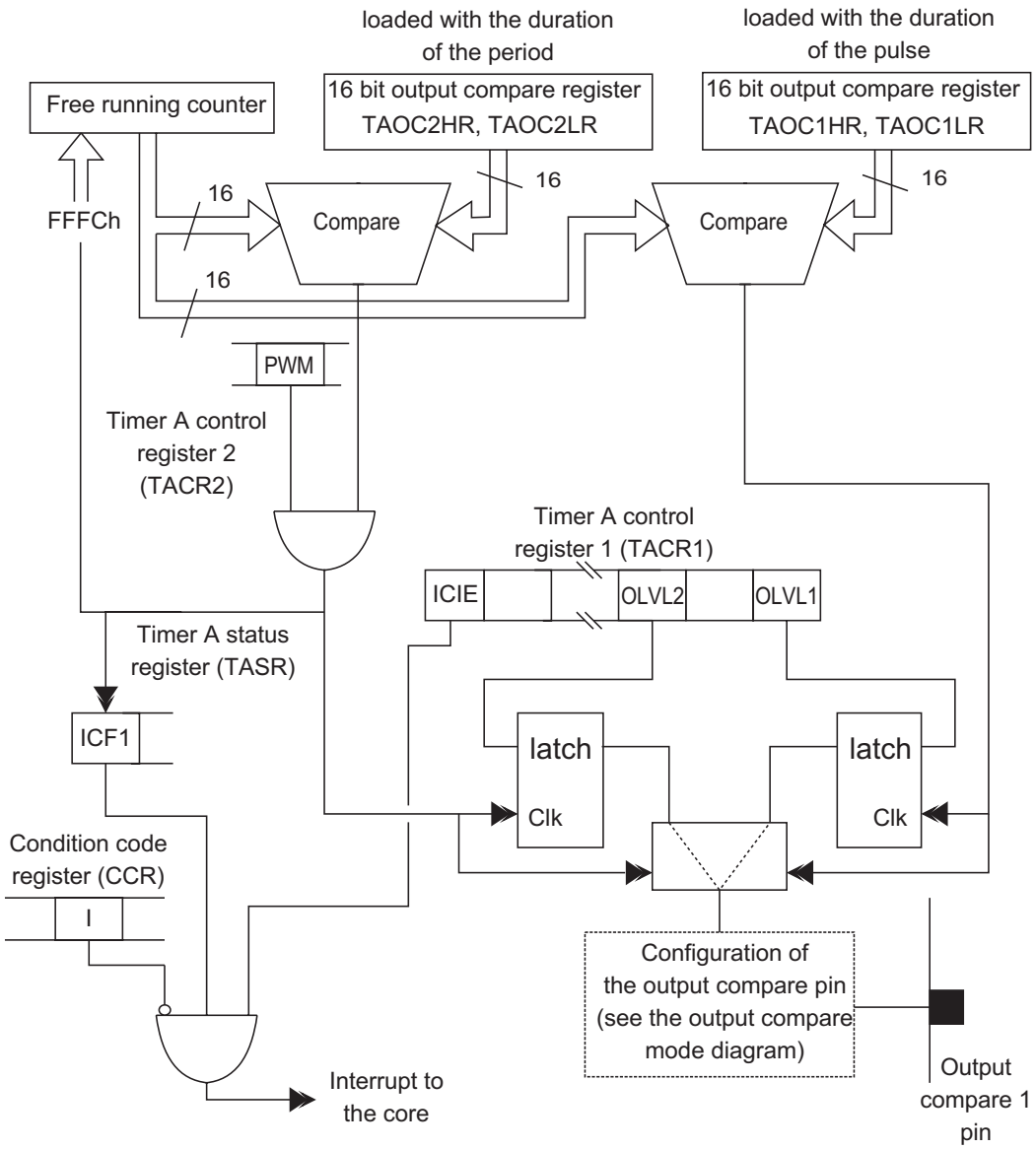
- The input capture event, corresponding to the leading edge, sets the ICF1 bit of TASR. This can trigger an interrupt if the ICIE mask bit is set. It must then be reset by software as explained above.
- The output compare 2 event, corresponding to the trailing edge, sets the OCF2 bit of TASR. This can trigger an interrupt if the OCIE mask bit is set. It must then be reset by software as explained above.

The OCF1 bit is never set in this mode. This allows the interrupt requests generated by these two events to be separately enabled or disabled.

5.5.6 Pulse-Width Modulation mode

Two output compare circuits of the same timer are involved simultaneously in Pulse-Width Modulation mode. This mode is selected by setting the PWM bit of the TACR2 register (or TBCR2 for Timer B). The PWM mode and the OPM mode are exclusive; if both selection bits are set at the same time, OPM mode is overridden by PWM mode.

In this mode, no external event resets the free-running timer; instead, the second compare circuit is used. Output Compare register 2 should be set to the value of the repetition period, converted in timer ticks, minus 4. Each time the free-running timer matches the Output Compare 2 register, the same events occur as an the input capture occurs in One Pulse Mode: output pin OCMP1_A is toggled, and the free-running counter is reset to FFFCh. Later, a successful match with the value in the TAOC1R register toggles the output back.



PWM mode and corresponding interrupt mechanism: diagram for timer A

05-PWM

The settings for this mode are performed as follows:

- Set the TAOC1HR-TAOC1LR pair to the number of ticks corresponding to the duration of the output pulse (this number depends on the clock frequency), minus 4.

- Set the TAOC2HR-TAOC2LR register pair to the number of ticks corresponding to the duration of the whole cycle (this number depends on the clock frequency), minus 4.
- Set the OLVL2 bit in the TACR1 register to the state required for the output pin for the duration of the pulse, and OLVL1 in TACR1 to the complement of this state to terminate the pulse.
- Set the PWM bit in TACR2 to enable the Pulse-Width Modulation mode.
- Set the PWM bit in TACR2 to enable the output pin.

The OCF1 and OCF2 bits are never set in this mode. However, the leading edge of the pulse, corresponding to the start of cycle, can generate an interrupt if desired. Actually, the Compare 2 event, as mentioned before, mimics a Capture 1 event. This sets the ICF1 bit in the TASR register. This can trigger an interrupt if the ICIE mask bit is set. It must then be reset by software as explained for Input Capture.

The physical Capture 1 input, ICAP1_A, is inhibited for this reason, but the other input pin, ICAP2_A remains active. This allows input captures to be performed while the timer is used in Pulse-Width Modulation mode. An example of this is given in the first application in Chapter 9. In this application, the timer generates periodic pulses while its frequency is controlled using a Phase-Locked Loop scheme to synchronize it with an external reference frequency.

The application described in Chapter 9 uses the timer in PWM mode with interrupt generation. The capture feature is also used at the same time to synchronize pulse generation with an external signal, using a Phase-Locked Loop. Another timer is used to produce an interrupt after a delay each time the timer is reinitialized for the program's needs.

5.6 ANALOG TO DIGITAL CONVERTER

5.6.1 Description

As a measuring device, the converter is specified by parameters that give the degree of accuracy of the conversion. These are:

- Input range: positive. Negative voltages are not converted.
- Type of conversion: ratiometric.
- Resolution: 8 bits, that is, 256 discrete values.
- Linearity: the conversion is guaranteed monotonic. This means that when the input voltage increases, the converted value either remains the same or increases, never decreases. The same applies when the voltage decreases.
- Accuracy: to within 1 LSB worst case, all causes of error included; typical 0.3 LSB. This means that for any voltage at the input, the conversion result does not differ from the expected result by more than 1/255 of the supply voltage in the worst case, or 0.12% typically.

- Conversion time: 64 clock cycles of the ADC. The ADC being fed with the core clock frequency.

These somewhat cryptic features actually mean the following simple facts:

The input voltage must remain positive or null; below zero volt, the conversion yields zero. In addition, the voltage at any input is limited to –0.3 V. If a signal ranges includes the zero, two solutions are available as described later in this chapter.

The ADC of the ST7 is a ratiometric converter, that is, it returns a binary number that expresses the ratio between the input voltage and the supply voltage. A input of zero volt (or lower) provides a binary result of zero; an input of V_{DD} (or more) provides a result of 255. The resolution is eight bits, which means that the converter distinguishes between 2^8 voltage values, that is 256 values.

Accuracy and linearity are important features since they determine whether a converter is suitable or not for a certain job. A discussion of this subject is given below.

The conversion time depends on the core clock frequency. This is a factor that must be taken into account when selecting the crystal frequency and the division rate, since in slow mode the clock to the converter is also slowed down.

5.6.2 Using the Analog to Digital Converter

Controlling the ADC is fairly simple, since it is controlled by a single register (ADCCSR).

Bit 5 of the CSR (ADON) is the on/off switch of the ADC. When off, it does not consume power, reducing the dissipation of the ST7. When switched on again, for the first 30 µsecs, conversions may be inaccurate.

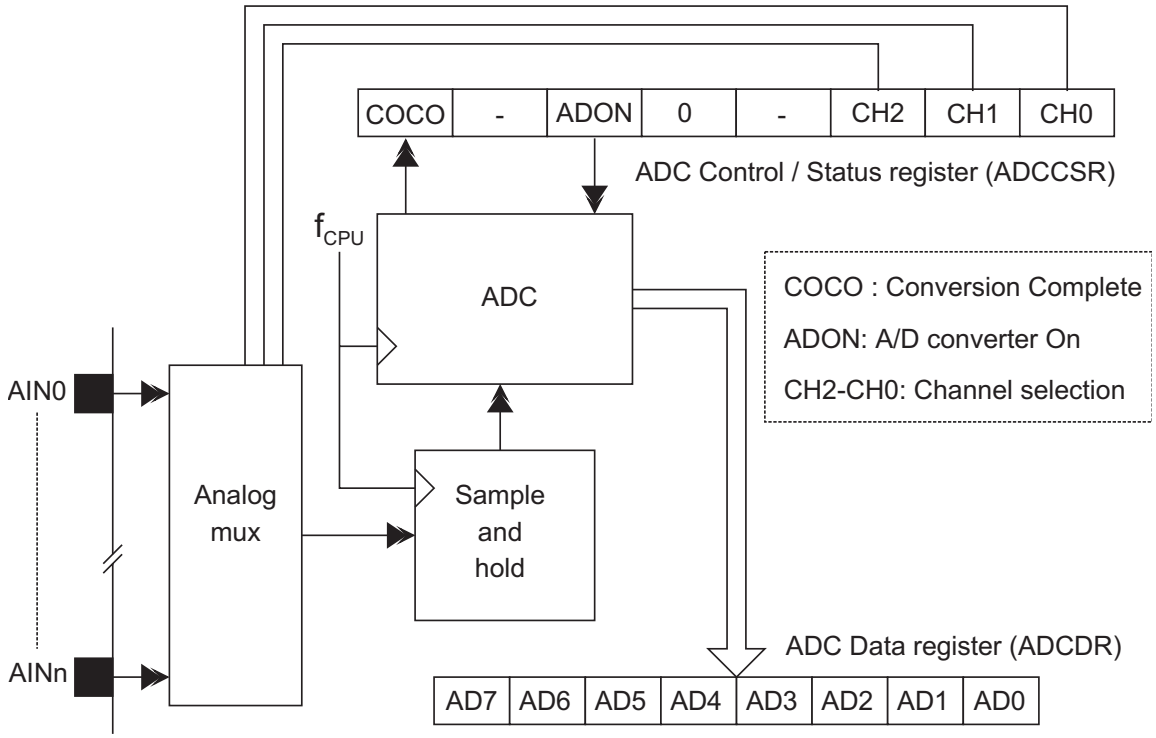
Bits 2 to 0 select the input pin whose value is to be read. The inputs to the ADC are actually pins taken from a parallel port. So the pins that are to be used as analog inputs must be configured as input, no pull-up, no interrupt (DDR=0, OR=0) to put them in high-impedance and avoid any disturbance on them.

Six analog inputs are available on the ST72251: PC0 to PC5

Eight analog input are available on the ST72311: PD0 to PD7.

Any write to the ADCCSR register stops the conversion in progress and starts a new one. When conversion is finished, bit 7 of the ADCCSR (COCO) is set to one meaning that data is available in the Data Register (ADCDR), and a new conversion starts. The converter thus continuously converts the input value; any read from the ADCDR will return a value that gives the voltage of the corresponding input, measured at a time no earlier than the conversion time specified above (2 speeds). If you do not need a time precision greater than this value, it is simplest just to leave the converter running continuously and read the value whenever you need it.

The COCO bit is reset when the ADCCSR register is written.



ADC block diagram

05-adc0

5.6.3 The problem of the converter's accuracy

Linearity and accuracy are different ways of expressing the same reality: the successive voltage steps that correspond to each of the binary values are not absolutely equally spaced (otherwise the converter would be perfect), leading to a conversion error. Depending on the type of work the converter is employed for, one way or the other is better for expressing the suitability of the converter.

If the converter is to be used as a measuring input, like measuring a temperature, a voltage, the level in a tank, to send it for information purposes, accuracy is the best expression. It indicates how much confidence one can have in the data.

If the converter is to be used as the feedback input for a servo loop of the second order, for example a positioning device using a DC motor and a feedback potentiometer, then linearity is the key factor. Not only must a servo loop be accurate (since this accuracy translates into an error expressed in mm in the positioning system) but also it must be stable. So, the parameters of the whole system must meet certain conditions summarized in the so-called Nyquist

criterion, that states the shape of the curve that relates the closed-loop gain with the phase shift. These parameters depend on the gain and phase shift of all of the components of the system, including the ADC. The local gain of the converter is the slope of the conversion curve. This value is given by the ratio between the actual height of the step and its theoretical height. This number is either greater or less than zero, depending on the point considered. This local gain multiplies the total loop gain, and thus may greatly affect it. Thus there may be some points where the gain is too high to meet the Nyquist criterion, leading to instability at these points.

Generally speaking, implementing a numeric servo loop requires very high resolution and linearity, combined with a short conversion time. This means an expensive converter, and will be outside the range of the ST7 ADC unless the application just needs a servo loop but does not need high performance.

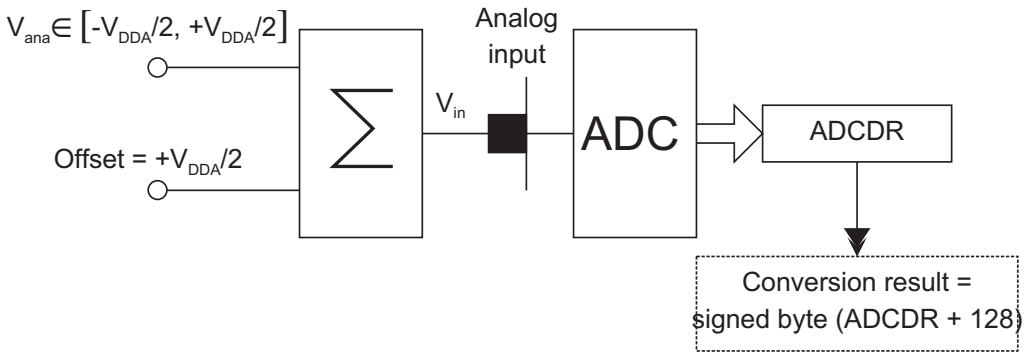
5.6.4 Using the ADC to convert positive and negative voltages; increasing its resolution

5.6.4.1 Measuring negative and positive voltages

There are two ways of handling both positive and negative voltages with a converter that can only handle positive voltages.

The first way is to amplify or attenuate the signal and to shift it by adding a fixed DC voltage so that when it varies within its whole range, the ADC is fed with a voltage between zero and V_{DD} (that is 5V in most cases). Then, when reading the ADCDR register as an unsigned value, subtracting the DC offset yields a signed number that is zero when the signal to read is zero.

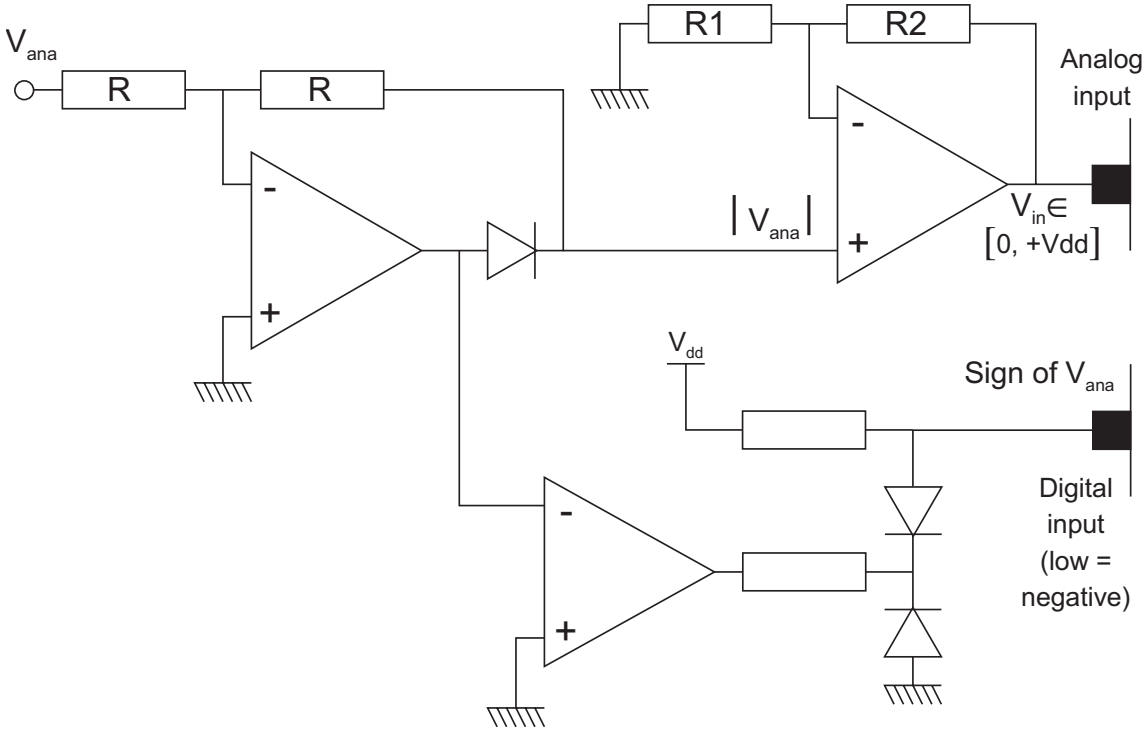
This method is simple, but it reduces the resolution since the whole signal range must fit the 256-value range of the converter.



Measuring relative voltages using an analog offset

05-adc1

The second method uses an absolute rectifier circuit that produces a voltage that is the absolute value of the input voltage. This circuit also provides a bit that indicates the sign of the input voltage. Thus the conversion of the whole range is increased by one extra bit that doubles the resolution.



Measuring signed voltages using an absolute rectifier

05-adc2

5.6.4.2 Increasing the resolution

The resolution of the converter is the number of different values of the input voltage that it can distinguish. This number depends on the actual number of discrete voltages that the converter can internally produce and compare to the input voltage. It would appear that it is not possible to change it since it is built into the silicon chip.

Luckily, there are ways of doing it from the outside. Let us examine the principle first, then a real application.

Let us assume that we feed the converter with a voltage V0. This voltage will lead to the conversion result Adc0. What does this mean? It means that the converter has seen a voltage ranging between that of the step Adc0 and that of the step Adc0 + 1. We do not know where this voltage lies within that range.

Let us now add to that voltage V0, a small voltage, that is equal to half the smallest step, that is,

$$\frac{Vdda}{255 \times 2}$$

If the voltage V0 is included within

$$\frac{Vdda \times Adc0}{255} \leq V0 \leq \frac{Vdda \times \left(Adc0 + \frac{1}{2}\right)}{255}$$

the converter will convert it as Adc0.

If the voltage V0 is included within

$$\frac{Vdda \times \left(Adc0 + \frac{1}{2}\right)}{255} \leq V0 \leq \frac{Vdda \times (Adc0 + 1)}{255}$$

the converter will convert it as Adc0 + 1.

We can now distinguish more finely between two values of the input voltage, as soon as they differ by more than

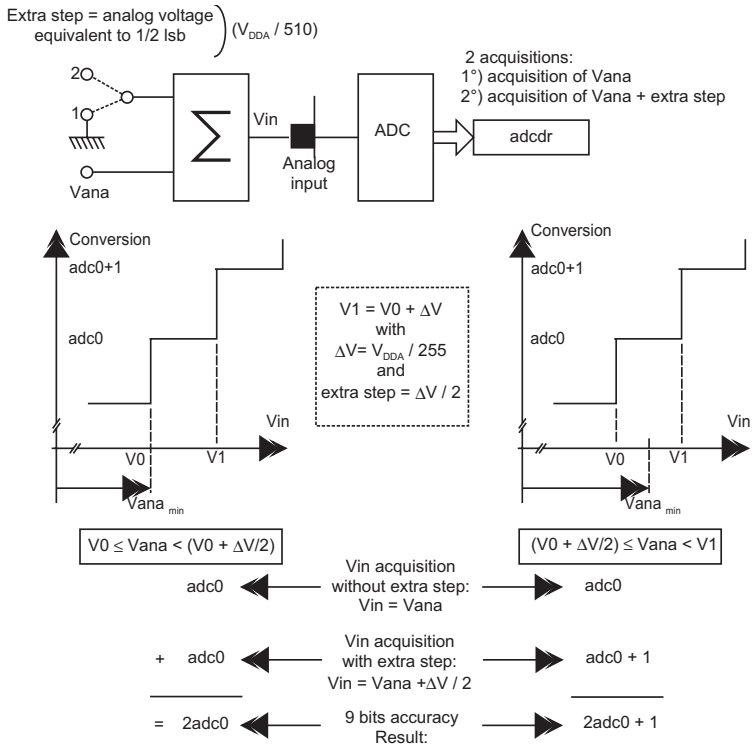
$$\frac{Vdda}{255 \times 2}$$

Let us exploit this by doing two successive conversions of the voltage V: the first time with V0 alone, the next time with

$$V0 + \frac{Vdda}{255 \times 2}$$

This will yield two results, Adc0 and Adc1. If we take the sum of these values, whatever the value of V, it will range from 0 + 0 = 0 to 255 + 255 = 510. This new result now has a resolution of 511 points, which is almost double the original resolution.

The figure below depicts this.



Example of a simplest scheme using an extra step signal to double the accuracy

05-adc3

Actually, this method is not very convenient, because the typical error of the converter is 0.3 LSB which is too high compared to the small voltage that we intend to use as an extra step. Also, the noise would make this increased resolution meaningless.

A practical way of increasing the converter resolution is to use the method of low-pass filtering of noisy signals.

Let us assume that we disturb the voltage we want to measure with a random voltage that has the following characteristics:

- The statistical average of the random voltage is zero, that is, is has a null D.C. component;
- Its average amplitude is about one LSB of the converter or more.

If we take a certain number of readings with the input voltage V0 fixed, but with a noise with the above characteristics added to it, and we compute the average, we shall find the same value as a single conversion with no noise, since the average of the noise voltage is zero. But if we look at the successive values that have been converted, they are not all equal because of the noise that has either added of subtracted a certain count to or from the ideal conversion result. If, instead of taking the mean of these values, we just add them, we get a number that ranges from 0 to n * 255 where n is the number of readings added together. Since the successive conversion results are not identical, all the values in the range are possible. For example, if we sum four readings, the result ranges to 4 * 255 = 1020. We can now distinguish 1021 different values of the input voltage.

This explanation is a bit too simple also, for the average of the successive values of the noise tends to zero for a large number of readings. If we only take a small number of readings, the sum of the readings will itself be affected by a noise. It is possible to calculate the number of readings that must be summed together to have that noise smaller than the resolution we expect to get.

5.6.4.3 Application Examples

The application described in Chapter 7 uses the converter to get a value derived from the position of a potentiometer. This is a convenient and cheap way of inputting an 8-bit value, also used in the second application, in Chapter 10, where three calibration values are input using three potentiometers.

In Chapter 9, the converter is used as a comparator to produce a logical signal depending on whether the analog input signal is above or below a predefined threshold.

An example of increasing the resolution using the summing method above is given in the second application, Chapter 10.

5.7 SERIAL PERIPHERAL INTERFACE

The Serial Peripheral Interface is a device that allows synchronous, bit serial communication between a microcontroller and a peripheral such as a serial-access memory, a Liquid Crystal Display module, etc.; or between two or more microcontrollers. This interface only requires three lines, thus saving pins for other purposes. The data is sent on a byte by byte basis. To perform the same function using a byte-wide parallel interface, at least ten lines are needed: eight for the data plus two for the handshake.

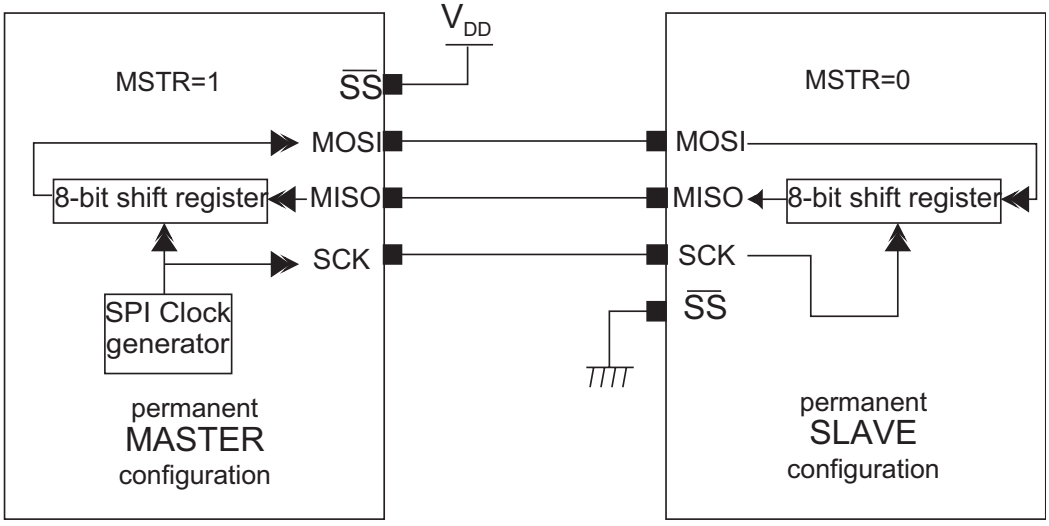
When connected to a peripheral, the SPI is configured as a master; when connected to another microcontroller, it is configured either as a master or as a slave.

The maximum bit rate of a master is 4 MHz with a 16-MHz crystal, or 250 kBytes per second. With this speed, serial transmission is just as fast as parallel transmission, since the instructions needed to put or get a byte through the channel take longer than the transmission.

Actually, transfer is bi-directional; at the same time that data is sent on one line, it is received from the other line; this allows full-duplex transmission at the rate mentioned above when two microcontrollers are connected together.

A supplementary pin, $\overline{SS}$, allows the SPI to be enabled when it is in slave mode. This allows a master to be connected to several slaves and to communicate with only one slave at a time. All the logic needed for multidrop communication is provided, including collision detection. This permits reliable interprocessor communication. When the SPI is in master mode, this pin must be set to a high level.

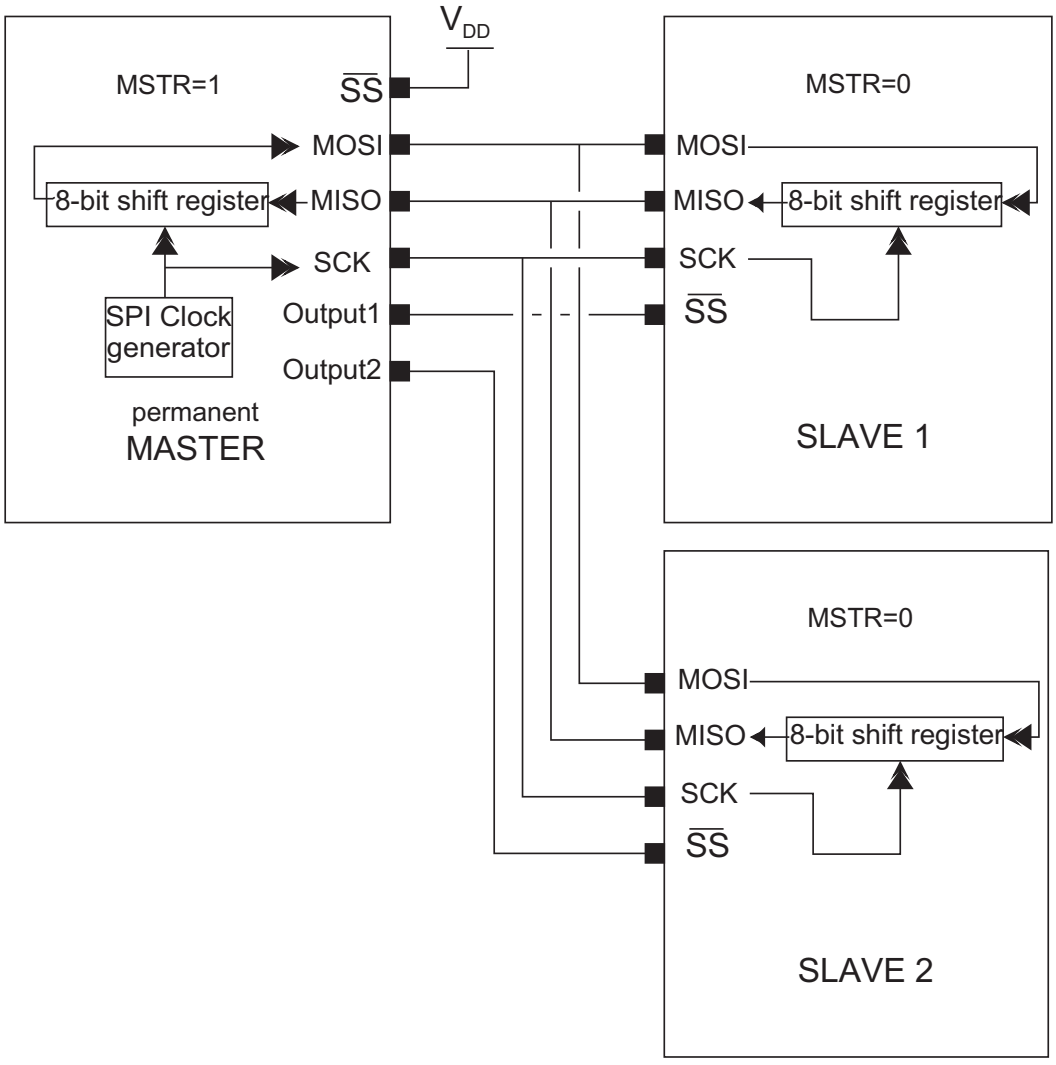
Here is an example of two microcontrollers connected together:



Simplest configuration for a dialogue with SPIs between two microcontrollers

05-spi1

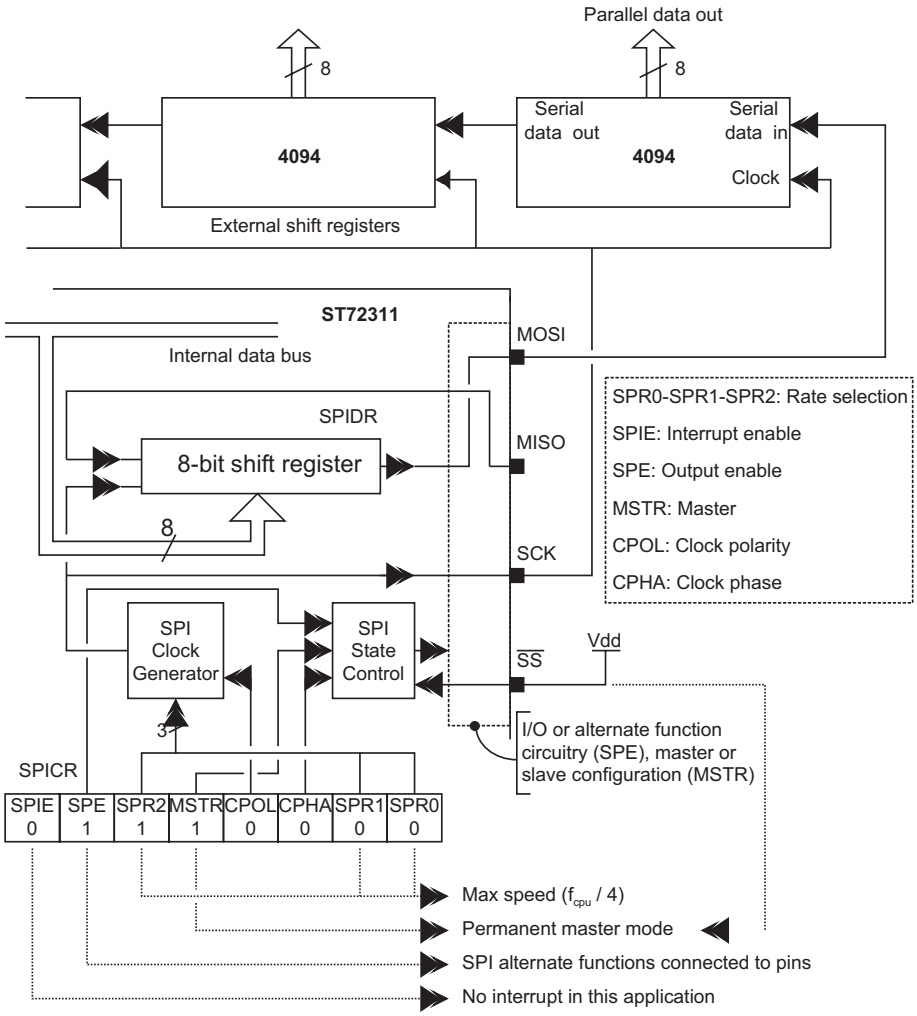
When one master controls two or more slaves, a method must be used to distinguish between the two slaves. Several methods are available. The one shown below uses the $\overline{SS}$ pin of each slave to control which slave is active at any given time. These pins are each driven by a separate output pin of the master controller:



Single master system for a dialogue with SPIs between three microcontrollers

05-spi2

An example of the use of the SPI to send data to a LCD display module is given in the second application, Chapter 10. The configuration used is shown in the figure below with the appropriate values in the registers.



Expanding the parallel outputs using the SPI and external shift registers

05-app10

The CPOL and CPHA bits in the control register allow you to select which clock edge is used externally as the active one (leading or trailing) and its polarity (low or high level after transmission).

5.8 SERIAL COMMUNICATION INTERFACE

The Serial Communication Interface is perhaps the most classical interface used when two systems are connected together. This is especially true when a small system is connected to a PC, either permanently, or temporarily, for instance for calibration, logging or maintenance.

The SCI differs from the SPI in several ways:

- The clock is not transmitted along with the data;
- The first data bit is preceded by a Start bit;
- The first data bit sent is the low-order bit;
- The last data bit is followed by a Stop bit.

5.8.1 Bit rate generator

The bit clock is derived from the CPU clock, divided by a user-selectable value. There are two ways of doing this:

For the most popular bit rates, the Baud Rate Register offers a choice of four prescaler values, and the output of the prescaler is further divided by two separate divisors that provide the receive bit rate and the transmit bit rate. The prescaler is driven by the core clock divided by 32.

This gives the following combinations (only the receive bit rate is considered here, because the transmit bit rate is produced exactly the same way):

For a 8 MHz core clock (16 MHz crystal):

	Prescaler value			
Divider value	**1**	**3**	**4**	**13**
1	250000	83333	62500	**19231**
2	125000	41667	31250	**9615**
4	62500	20833	15625	**4808**
8	31250	10417	7813	**2404**
16	15625	5208	3906	**1202**
32	7813	2604	1953	**601**
64	3906	1302	977	**300**
128	1953	651	488	**150**

With another core clock, we would get different values. The values in bold are the most commonly used. Please note that they are not exact; however, asynchronous serial transmission is by nature tolerant of bit rate errors of up to 4%. The error here is only 0.16%.

The division ratio is selected by bits in the Baud Rate Register, with two bits for the prescaler value, then three bits for the receive divisor, and three more bits for the transmit divisor.

If no combination of the core clock and the division ratio provided by the BRR fits the your requirements, you can specify a value between 1 and 255 as a prescaler for the receiver by writing that value in the Extended Receiver Prescaler Register. The core clock is then divided by 16, and by this division ratio. If the ERPR is set to zero, then the selections of the BRR above apply.

Similarly, the transmitter bit rate can be fine-tuned using the Extended Transmit Prescaler Register.

5.8.2 Send and receive mechanism

The data to be sent and the data that is received are both put in the Data Register. When this register is written, it starts the transmit process. When a word is received, it is copied to that register where it can be read.

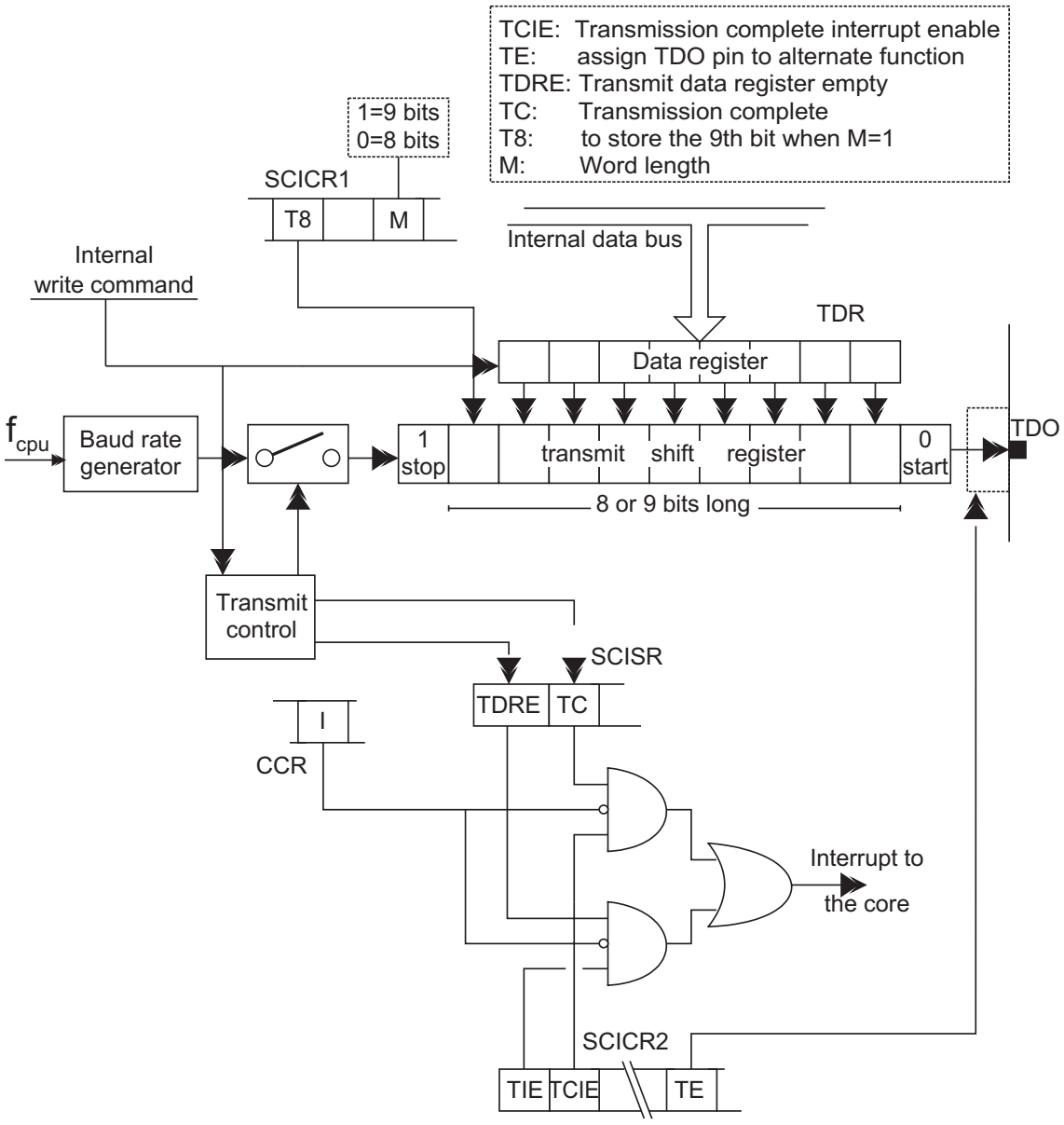
The word length is set in the Control Register 1 by the M bit. If the M bit is cleared, the word length is 8 bits; otherwise, it is 9 bits. The ninth bit, when received, is copied to the R8 bit of that same register. To transmit a 9-bit word, the first 8 bits are taken from the Data Register (least-significant bit first) and the value of the T8 bit of the CR1 register is used as the ninth bit.

No automatic parity generation or checking is provided. If needed, parity may be calculated by software, then copied to either bit 7 of the byte to send, if the word length is 7 bits plus parity, or to bit T8, if the word length is 8 bits plus parity.

The serial transmission is straightforward: if the transmit shift register is empty, the data byte written to the Transmit Data Register is copied to it. The serial sending process starts then, by sending a zero bit (the start bit), then the byte to transmit, LSB first, then the T8 bit if the word length is set to 9 bits, then a one bit (the stop bit). The transmission is then complete.

As soon as the Transmit Data Register is empty, the TDRE bit in the Status Register is set. When the transmission is complete (when the shift register is empty), the TC bit is set. These bits are cleared by first reading the SCISR register then accessing the SCITDR register.

The SCITDR acts as a buffer to allow a continuous data flow on the serial line, by reacting to the interrupts that are generated when TDRE is set. This allows the core to supply the next character in the time needed to transmit a character (about 500 µs at 19200 bits per second).

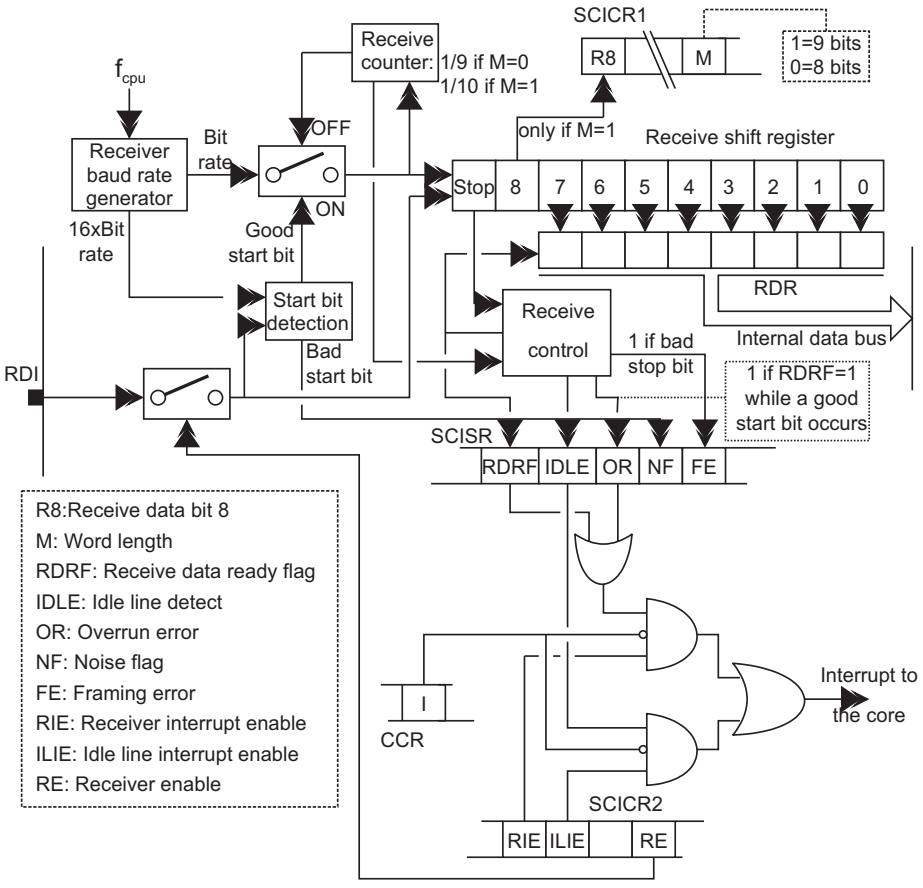


SCI simplified transmit bock diagram

05-SCI1

The receive mechanism is a bit more complicated. The receive clock must first be set to the same frequency as the transmit clock of the source of the serial data to be received. Since the start bit is used to resynchronize the clocks, the clock frequency has a fairly wide tolerance since it is allowed to shift by a half-bit period at the end of the transmission, that is, 11 bits. This gives a tolerance of about 5%, which is not a problem to achieve.

The resynchronization is the most difficult thing. It uses a clock a 16 times the bit rate, and the state of the receive data pin is checked at each period of that clock. When a falling edge is detected, a mechanism takes that time plus half a bit period as the reference time for the sampling of all subsequent bits. The start bit is also checked to be a zero at the sampling time. It is not, it is considered a false start bit. The NF bit is set in the SCISR, and the receiving sequence is not initiated. All successive bits are shifted into the receive shift register. When the stop bit arrives, it is also checked for a one. If it is a zero, the FE bit is set in the SCISR. The RDRF bit is set when the reception is complete. It can generate an interrupt, so that the received character may be picked up.



SCI simplified receive bock diagram

05-sci2

5.8.3 Status register

The Status Register includes the following bits that show the current status of the SCI:

- The TDRE and TC bits, if set, indicate that a new character may be supplied to continue the transmission. The difference between these bits comes from the fact the SCI has a buffer before the transmit shift register. The TDRE bit indicates that the buffer is empty, but the shift register may not be empty. TC indicates that the shift register is empty, which also implies that the buffer is empty. Depending on what you want to do, either or both of these bits may be taken into account: TDRE indicates that the next character to send may now be supplied; TC indicates that the last character is sent.
- The RDRF bit indicates that a character has been received.
- The IDLE bit goes to one when a full character time or more has elapsed since the last character was received, indicating that the incoming character flow is suspended.
- The next three bits show that an error condition has been detected. The OR bit (OverRun) indicates that the last-but-one character that remained unread in the Data Register has been overwritten by the last character received: it is thus lost. Each bit is sampled three times, if the three sample are not the same, the NF bit (Noise Flag) is set and the bit value is determined according to the 2 to 1 majority rule. The FE bit (Framing Error) indicates that a proper stop bit was not present at the end of the character. The interpretation of these error conditions is not necessarily pertinent; however, the occurrence of any of these errors tends to mean there is a transmission problem and that the data transfer is not reliable. The character received in the Data Register is probably incorrect.

There is one case where a Framing Error is detected and where this condition is expected. It is the case of a Break condition. A Break is a state of the line where it is in its active state (“Mark”, or zero) for at least one character period. This may indicate that the line has been disconnected. It is also possible that the transmitter puts a normally connected line in Break condition. This may be used to signal a particular event and it is up to the system designer to decide which event.

5.8.4 Control Register 2

The Control Register 2 contains the following bits:

- TIE and TCIE, if set, enable an interrupt request when the TDRE or TC bits in the Status Register are set, respectively.
- RIE, if set, enables an interrupt when the RDRF bit in the Status Register is set.
- ILIE, if set, enables an interrupt when the IDLE bit in the Status Register is set.
- TE and RE enable the transmitter and the receiver respectively, and change the appropriate port pins to Serial Output and Serial Input.

- RWU, when set, places the receiver in a mode where the RDRF bit is not set and a receive interrupt is not generated, even when characters are received. The SCI exits this sleep state only when one of the following events occur: the ninth bit is a one, with the word length set to 9; or when an idle line condition is detected. Which of these events wakes up the SCI depends on the state of the WAKE bit in Control Register 1. If cleared, the Idle line condition wakes the SCI; if set, the ninth bit set does it.
- SBRK sets the transmit line to the Break condition. The line remains in this condition until SBRK is cleared. No characters can be sent while in Break condition.

5.8.5 Using the Wake-Up feature in a multiprocessor system

Waking-up the SCI when the ninth bit is set allows you to build a network of microcontrollers, all connected to the same line. Then, by convention, if a device sends a character with that bit set, it can be considered by all microcontrollers as an address character. If the value of the address received does not match the local address, a microcontroller has nothing to do, since it will only be interrupted when addresses are received. If the address matches the local address, the microcontroller may clear its RWU bit, thus switching the SCI to normal mode. It then receives all the characters. Next time an address is received, and if that address is not its own, the microcontroller just sets its RWU bit and from that time on, it is not disturbed by the traffic on the line.

5.8.6 Handling the interrupts

All the interrupt causes of the SCI share the same interrupt vector. Thus, the same interrupt service routine is used for all causes, and the interrupt service routine must start by testing the Status Register bits to know which event caused the interrupt.

For this reason, and as ususal for most of the ST7 peripherals, once an interrupt has been accepted and the interrupt service routine is started, the interrupt request must be cleared by program. The way to do it depends on the interrupt considered.

- The receive interrupt request bit is cleared by a read of the Status Register followed by a read of the Data Register. Since these actions must anyway be done in the program, first testing which bit is set in the Status Register and then reading the received character, the clearing of the interrupt request is transparent to the programmer.
- The transmit interrupt request bit (either TD or TDRE) is cleared when the Status Register is read and the Data Register is written. As above, this is transparent to the programmer.
- The IDLE condition interrupt request bit is only reset by resetting the RE (Receive Enable) bit, even temporarily.
- The error bits OR, NF and FE are also only reset by resetting RE.

Using interrupt-driven software to handle the SCI is generally a good solution. A received character or string is written to a buffer, then the main program is informed that something has

been received by a flag set by the interrupt service routine. Conversely, if the main program wants to send data, it prepares the character or string to send into a buffer in memory, then starts the transmission by enabling the transmit interrupt. This will make the interrupt mechanism send the characters one at a time, an interrupt occurring each time the SCI has sent a character. The sending is terminated either by exhausting a character counter or encountering a terminator at the end of the buffer. An example of this type of handling is described in the second application, Chapter 10.